U.S. Department
of Commerce

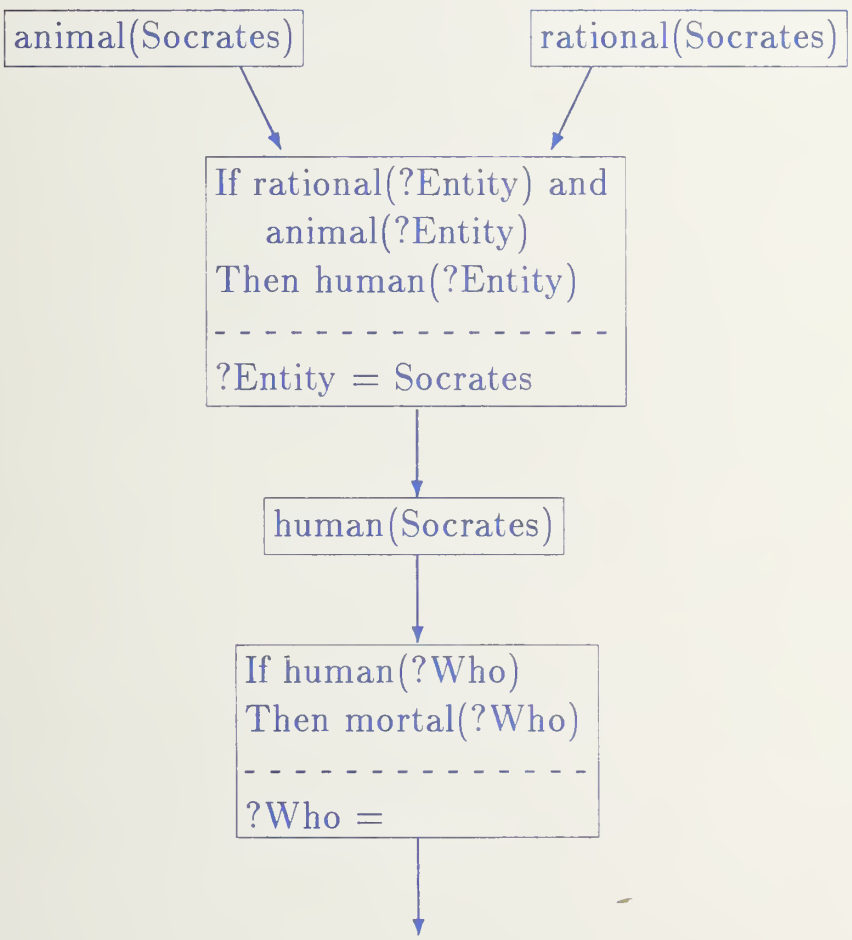
National Bureau
of Standards

Computer Science and Technology

NBS Special Publication 500-145

Programming Languages for Knowledge-Based Systems

John V. Cugini



The National Bureau of Standards[1] was established by an act of Congress on March 3, 1901. The Bureau's overall goal is to strengthen and advance the nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau's technical work is performed by the National Measurement Laboratory, the National Engineering Laboratory, the Institute for Computer Sciences and Technology, and the Institute for Materials Science and Engineering.

The National Measurement Laboratory

Provides the national system of physical and chemical measurement; coordinates the system with measurement systems of other nations and furnishes essential services leading to accurate and uniform physical and chemical measurement throughout the Nation's scientific community, industry, and commerce; provides advisory and research services to other Government agencies; conducts physical and chemical research; develops, produces, and distributes Standard Reference Materials; and provides calibration services. The Laboratory consists of the following centers:

- Basic Standards[2]
- Radiation Research
- Chemical Physics
- Analytical Chemistry

The National Engineering Laboratory

Provides technology and technical services to the public and private sectors to address national needs and to solve national problems; conducts research in engineering and applied science in support of these efforts; builds and maintains competence in the necessary disciplines required to carry out this research and technical service; develops engineering data and measurement capabilities; provides engineering measurement traceability services; develops test methods and proposes engineering standards and code changes; develops and proposes new engineering practices; and develops and improves mechanisms to transfer results of its research to the ultimate user. The Laboratory consists of the following centers:

- Applied Mathematics
- Electronics and Electrical Engineering[2]
- Manufacturing Engineering
- Building Technology
- Fire Research
- Chemical Engineering[2]

The Institute for Computer Sciences and Technology

Conducts research and provides scientific and technical services to aid Federal agencies in the selection, acquisition, application, and use of computer technology to improve effectiveness and economy in Government operations in accordance with Public Law 89-306 (40 U.S.C. 759), relevant Executive Orders, and other directives; carries out this mission by managing the Federal Information Processing Standards Program, developing Federal ADP standards guidelines, and managing Federal participation in ADP voluntary standardization activities; provides scientific and technological advisory services and assistance to Federal agencies; and provides the technical foundation for computer-related policies of the Federal Government. The Institute consists of the following centers:

- Programming Science and Technology
- Computer Systems Engineering

The Institute for Materials Science and Engineering

Conducts research and provides measurements, data, standards, reference materials, quantitative understanding and other technical information fundamental to the processing, structure, properties and performance of materials; addresses the scientific basis for new advanced materials technologies; plans research around cross-country scientific themes such as nondestructive evaluation and phase diagram development; oversees Bureau-wide technical programs in nuclear reactor radiation research and nondestructive evaluation; and broadly disseminates generic technical information resulting from its programs. The Institute consists of the following Divisions:

- Ceramics
- Fracture and Deformation [3]
- Polymers
- Metallurgy
- Reactor Radiation

[1]Headquarters and Laboratories at Gaithersburg, MD, unless otherwise noted; mailing address Gaithersburg, MD 20899.
[2]Some divisions within the center are located at Boulder, CO 80303.
[3]Located at Boulder, CO, with some elements at Gaithersburg, MD.

Computer Science and Technology

NBS Special Publication 500-145

Programming Languages for Knowledge-Based Systems

John V. Cugini

Center for Programming Science and Technology
Institute for Computer Sciences and Technology
National Bureau of Standards
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Malcolm Baldrige, Secretary

National Bureau of Standards
Ernest Ambler, Director

Issued February 1987

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

National Bureau of Standards Special Publication 500-145
Natl. Bur. Stand. (U.S.), Spec. Publ. 500-145, 79 pages (Feb. 1987)
CODEN: XNBSAV

Library of Congress Catalog Card Number: 86-600602

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1987

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington DC 20402

Programming Languages for Knowledge-Based Systems

by

John V. Cugini
Institute for Computer Sciences and Technology
National Bureau of Standards

Abstract

Knowledge-Based Systems (KBS) represent a new software methodology which can broaden the scope of computer applications. When developing such software at the programming level, symbolic languages offer features to the programmer not provided by traditional procedural languages. The three most widespread symbolic languages are Lisp, Prolog, and OPS5. An abstract model for a basic KBS and associated terminology is described. This provides a framework for evaluation of the languages. There are several criteria by which one may assess the relative merits of these languages for a given knowledge-based application. Some are related to the languages' expressiveness for typical KBS techniques, others to the user's requirements. An extensive set of these criteria is discussed, and the languages are evaluated in light of them. While Lisp offers more features for general-purpose and symbolic computing, it does not offer direct support for the derivation process. OPS5 and Prolog have features especially designed for KBS, but lack many common general-purpose constructs.

Key words: expert systems, knowledge-based systems, Lisp, OPS5, programming languages, Prolog.

Contents

List of Tables

List of Figures

1 Introduction

Knowledge-Based Systems (KBS) offer a new way of thinking about programming. Some important applications which cannot easily be programmed in the traditional algorithmic manner become quite tractable with this technique. Of course, there are many applications for which KBS offers no advantage over traditional practice, and others which are exceedingly difficult with any known approach. The information in this report, together with supplementary information from the Bibliography, should help users determine when to employ KBS techniques.

1.1 Purpose

The main purpose of this guide is to assess the relative merits of three programming languages, Lisp, Prolog, and OPS5, with respect to their suitability for developing a KBS. Most programmers do not as yet have extensive experience in the implementation of a KBS; the goal is to help them to make a wise and informed choice of language.

1.2 Scope

This guide describes and compares the features of three programming languages: Lisp, Prolog, and OPS5. They are analyzed primarily with regard to the processing model they provide for the implementation of a KBS. These three are the most commonly-used *symbolic* languages. A symbolic language is one which is designed for the manipulation of symbolic data, as opposed to such conventional data types as numbers and character strings. It is this ability which makes these languages appropriate for implementing a KBS, and distinguishes them from more conventional languages such as COBOL or Ada. A more detailed discussion of symbols as a data type may be found in section 4.5.1.

While our focus will be on the special attributes of these languages, their conventional features will also be covered. As in the case of Lisp, these may be quite extensive.

This guide is *not* meant to supply the following:

- Any information regarding non-symbolic Artificial Intelligence (AI) applications, such as vision or robotics
- A survey of KBS applications
- A fully detailed and general introduction to KBS
- A tutorial for any of the languages analyzed herein

- Any information concerning software development aids (other than the three programming languages themselves) for KBS's, such as so-called "expert system tools"
- A comparison of symbolic languages *per se* vs. conventional languages [Cugi84]

Readers interested in these topics may find useful material in the Bibliography.

1.3 Methodology

We will approach the subject of KBS languages by proceeding from the more abstract to the more concrete aspects. Section 2 covers the basic concepts of KBS in general, without regard to language issues. Thus we will have a reasonably neutral framework, within which to describe and evaluate the various language features.

When selecting a language, users should consider two questions: what the language provides and what they need. Sections 3 and 4 discuss the languages themselves and section 5 addresses the requirements imposed by the application and by the computing resources available at the user's installation.

Section 3 is devoted to the central issue of this report: how and how well the syntactic and semantic features of the languages support KBS programming techniques. Section 4, by contrast, covers their conventional features, i.e, those not peculiar to the implementation of a KBS.

A programming language provides two sorts of benefit: the logical power of the language itself, and the incidental effects that arise from its pattern of implementation. Section 3 and the early part of section 4 address the former issue, e.g., what data types are provided, what operations are built in, and so forth. The latter part of section 4 covers what we might call the extrinsic features of a language, e.g., how well it is standardized, the availability of software, etc.

Having surveyed what the languages have to offer, we then look at user requirements in section 5, and discuss how the language features support those requirements. Finally, section 6 summarizes the salient characteristics of each language and presents some general conclusions about their applicability.

The final methodological issue is which version of each language under study is to be evaluated. OPS5 and Prolog are of comparatively recent invention and are still evolving. We will *not* undertake to survey all features offered by various implementations, but will restrict ourselves to the *de facto* core of these languages, as of 1987. For Prolog, we will rely mainly on [Pere84], for OPS5, on [Forg81]. In the case of Lisp, fortunately, there has been an effort to reverse a trend towards divergent implementation, resulting in [Stee84], and we shall adopt that as our reference. For more information concerning standardization issues, see section 4.10.

2 KBS Concepts

In this section, we shall explain the concepts and terminology ordinarily associated with KBS's. Let us begin by analyzing the term "knowledge-based" itself. In what sense are KBS's "based" on "knowledge" that conventional systems are not? After all, even the most typical COBOL payroll program reflects a great deal of knowledge about a company's personnel system (e.g., policies regarding overtime, leave, pay status, etc.) and brings that knowledge to bear in solving a problem. As another example, a personnel database certainly captures knowledge about a company's employees.

The Knowledge Base. The distinguishing feature of KBS's is that the way knowledge is represented is in relatively close correspondence to the way in which it might be conceived by a human expert. The knowledge is not highly encoded in some specialized format, suitable mainly for machine processing. The resulting *knowledge base* (KB) then becomes a resource which can be used to develop a variety of related applications. A KB may include not only simple "data-like" knowledge, but also complex procedural knowledge (how to *transform* data). Further, procedural knowledge can be added to the base incrementally, as one might add data to a database, unlike the careful interweaving of such knowledge into an algorithm normally associated with program maintenance.

Contrast to Conventional Techniques. While a COBOL program may reflect knowledge about how to transform data, that procedural knowledge is implicitly and inflexibly encoded in its algorithm. The same design knowledge used to develop the algorithm is not directly available for other related purposes. Databases capture knowledge in a flexible application-independent way, but typically have no facilities for expressing complex procedural knowledge which may be used to develop new data from that already expressed explicitly. Of course database systems may provide certain standard operations for transforming data, such as *project* and *join,* but these are essentially syntactic transformations, applicable to any database. A KBS (or a procedural language) transforms data according to the specific rules for the domain. Thus, these rules capture semantic information about that domain.

Value of KBS Techniques. The salient characteristic of KBS's, then, is that they allow comparatively direct and explicit encoding of both data and procedural knowledge. Thus, they combine the power and expressiveness of procedural languages with the generality and flexibility of database systems. We are describing, of course, the ideal case. Not all KBS's actually offer the perfect modularity of knowledge described above. Further, even in such a system, the representation of the knowledge "nuggets" is a complex design issue, and strongly affects the power and efficiency of the system. Nonetheless, it is fair to say that

KBS methods approach the ideal more closely than conventional practice. See [Bobr86] for a more extensive discussion of the applicability of KBS techniques.

2.1 System-level Concepts

We may distinguish: 1) KBS applications, 2) KBS techniques, and 3) KBS-oriented languages. In this report we are mainly concerned with the relationship between 2) and 3), but their importance for the development of information systems becomes evident only in context with 1).

2.1.1 KBS applications

A *KBS application* is any function or operation whose software implementation employs KBS techniques as the major structuring mechanism. Typically, such applications are those for which the number of decisions to be made is rather large, and the order in which decisions should be made is unpredictable. When the order of decisions needed for an application is known in advance, ordinary algorithmic techniques usually suffice. An important class of KBS applications is that of so-called expert systems. We shall assume, as its name suggests, that an expert system is one whose major function is to simulate the behavior of a human expert in some domain of interest, such as medical diagnosis or circuit design. Thus, the term "expert system" refers to the external behavior of a program, whereas "KBS" adverts to its internal structure.

2.1.2 KBS techniques

KBS techniques are a set of logical constructs in terms of which one may understand the execution of a program. By analogy, we often speak of "structured programming techniques" by which we mean such things as the if-then-else construct, the do-while, modularity, etc. Similarly, KBS techniques include several software concepts which serve to describe the behavior of a program at a somewhat abstract level. Some common examples are rules and pattern-matching; these and others will be discussed below in detail.

2.1.3 KBS languages

Finally, a *KBS-oriented programming language* (KBS language, for short) is one which provides support for KBS techniques. Of course, this support may be more or less direct; one could, with enough effort, use KBS techniques in assembler language. We shall consider only those languages which have features especially designed to support KBS techniques.

2.2 Computational Concepts

Let us examine a computational model which is helpful in understanding the languages under study. This model is not meant to be definitive of KBS's in general, but rather to give us a vocabulary with which to describe these languages.

2.2.1 Facts and Rules

First, we need the notions of a *fact* and a *rule*. A fact is a statement about some particular individuals within the realm of discourse, for example: "John has brown eyes," "Maude's temperature is 102.3° Fahrenheit," or "Suzy's brothers are Richard, Stephen, Paul, and Joseph."

A rule is a statement about some general relationships within the domain that allows the derivation of new facts from old, e.g., "People with brown eyes are very conscientious" or "A person whose temperature is above 100.0° Fahrenheit is sick with the flu." It is often helpful to think of a rule as a conditional statement, with an *if-part* (also called the *antecedent*) and a *then-part* (or *consequent*). The first example above, for instance, might be re-cast as: "**If** a person has brown eyes, **then** that person is very conscientious."

Broadly speaking, then, a KBS is essentially concerned with the logical relationships among facts. This relationship is articulated by *applying* some relevant rule to a set of given facts and deriving some new facts as a result. Of course these new facts can then themselves serve as the basis for derivation of still other facts. A rule-application consists of the general rule itself and the *bindings* (or substitutions) of particular terms to general terms which connect it to the facts in question. For instance, given the examples of facts and rules in the preceding paragraph, we might conclude that "Maude has the flu." The rule involved is, of course, "A person whose temperature is above 100.0° Fahrenheit is sick with the flu" and the bindings involve substituting "Maude" for the general term "a person" and "102.3° Fahrenheit" for "whose temperature".

2.2.2 Formal Representation of Facts and Rules

Abstract Model. Typically, what we have been calling general terms are represented in KBS languages by *variables* (symbols which may represent different values at different times), and particular terms by *constants* (which can represent only a single value). Facts and rules are represented using structures built up from these atomic (indivisible) terms. Two of the most common structures are *lists* of elements, which provide a way of talking about several things collectively, and *predicates*, which represent properties of, or relations among, the objects or values of interest. These objects are called the *arguments* of the predicate. We say that the predicate is true of the arguments. Each argument may be labeled by a name, called an *attribute* or *role*, to describe its relationship to the predicate

and other arguments. As might be expected, these structuring mechanisms are quite general and can be applied to themselves, as well as to simple constants and variables, e.g., the elements of a list can themselves be lists or predicates.

Table 1: Example of Representation of Facts and Rules

	English	Formal
fact:	Maude's temperature is 102.3° Fahrenheit.	temp(person:Maude, value:102.3)
rule:	A person whose temperature is above 100.0° Fahrenheit is sick with the flu.	if temp(person:?Who, value:?What) and greater(num1:?What, num2:100.0) then sick(person:?Who, disease:flu)
fact:	Maude has the flu.	sick(person:Maude, disease:flu)
fact:	Suzy's brothers are Richard, Stephen, Paul, and Joseph.	brothers-of(person:Suzy, brothers:[Richard, Stephen, Paul, Joseph])

Constants - Maude, flu, 102.3, 100.0, Suzy, Richard, Stephen, Paul, Joseph
Variables - ?Who, ?What
Predicates - temp, greater, sick, brothers-of
List - [Richard, Stephen, Paul, Joseph]
Arguments - Maude, flu, 102.3, 100.0, Suzy, [Richard, Stephen, Paul, Joseph], ?Who, ?What
Roles - person, value, num1, num2, disease, brothers

Table 1 illustrates a possible formal representation of some facts and rules. In the table, the second fact is derived from the first by applying the rule, and binding the variable ?Who to Maude, and ?What to 102.3. We are assuming that the system can derive certain facts for us, in this case, that 102.3 is greater than 100.0. Notice that while a rule may contain variables and constants, a fact contains only constants.

Design of Knowledge Representation. It should be stressed that the way in which a set of facts or rules is formally representated is a very subtle and far-reaching design issue. For instance, "Maude has the flu" might be encoded as: has-flu(person:Maude), i.e., as a predicate with a single argument, rather than as the two-argument predicate above. The reader should not suppose that, given our abstract model, the representations shown in the examples are a uniquely correct way of capturing the information.

One very basic distinction in knowledge representation is between the *object-oriented* and the *property-oriented* approach. One short example must suffice: let us say we wish to encode the knowledge that John has red hair and that he is 190 cm. tall. Given our predicate notation above, either of the following is plausible:

Property-oriented	Object-oriented
hair-color(name:John, color:red)	person(name:John,
height-is(name:John, height:190)	hair-color:red,
	height-is:190)

As can be seen, such questions as whether a term acts as a predicate or as a role depend on how the facts are represented.

Property-oriented representation tends to fragment the knowledge into small independent pieces. It is associated with a *logic-based* approach to KBS. The predicates typically correspond to English verbs or adjectives.

Object-oriented representation tends to gather up all the information concerning an entity into one bundle. The *frame-based* model of KBS emphasizes object-oriented representation. The predicates usually correspond to common nouns in English which denote the type of entity being represented.

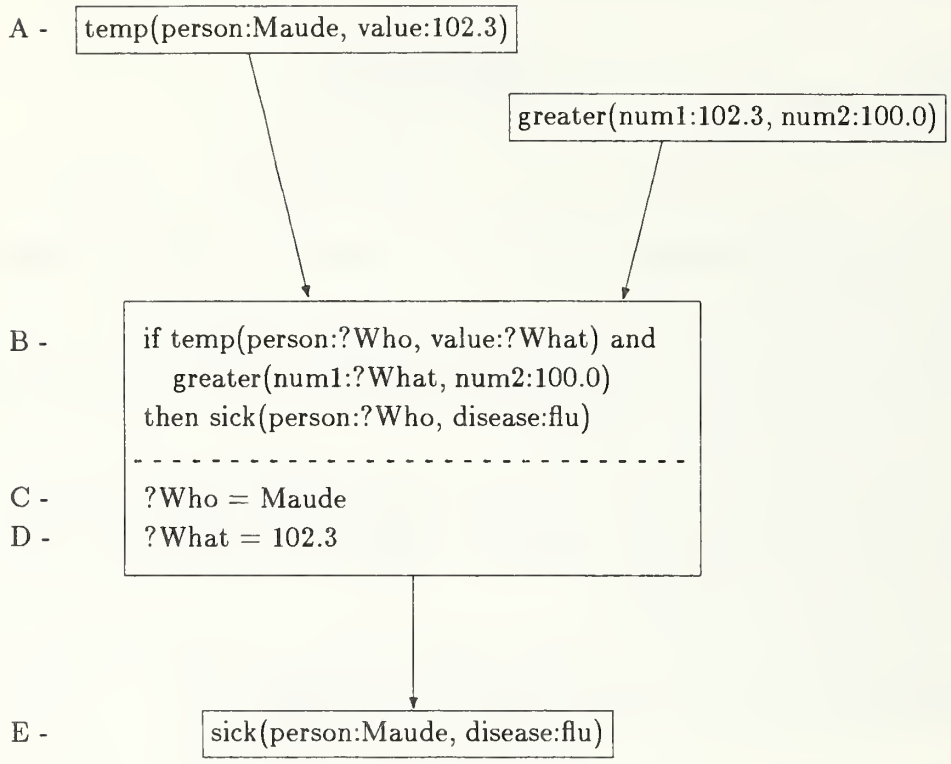
We shall not attempt an evaluation of the relative merits of these approaches in this report. Interested readers should refer to [IEEE83] and [Fike85].

2.2.3 Digraph Model of Derivation

We may represent the derivation process more formally by a directed graph (digraph). Each node represents either a fact or the successful application of a rule. The directed edges connecting the nodes represent the flow of derivation. Let us look at the digraph (see Figure 1) for the example we have been using. A and E are labels for the two facts, B is the rule part of the rule-application (above the dashed line), and C and D are the bindings in the rule-application.

It is natural to think of each node as having a set of predecessors (those nodes from which there is an edge to the node in question), and successors. Then, we may interpret the graph relationships as shown in Table 2. Facts are adjacent only to rule-applications and vice-versa, i.e., no two facts are adjacent, nor are any two rule-applications. All rule-applications have predecessors and successors; it does not make sense to speak of a rule successfully being applied to nothing, or generating nothing. Some facts, however, may have no predecessors: these are the facts, such as A in the figure, *given* to the system from some external collection of data. It is possible, of course, for a fact to be present by virtue of several (redundant) reasons: it could be given and also derived from a rule-application, or even from several applications. Such a fact might have several predecessors. Also, some

Figure 1: A Small Derivation Digraph



facts, such as E, may have no successors, either because no rule was applied to them, or because they are *goals*.

Goals are facts which, according to some specification, constitute a final answer or result, presumably to be reported to some external destination. Given facts and goal facts are an important part of the system's interface with the user.

Table 2: Interpretation of Digraph

	Predecessors	Successors
of a fact:	Rule-application(s) which generated it	Rule-application(s) using it
of a rule-application:	Facts used by this application	Facts generated by this application

The digraph is to be thought of as modeling a completed (successful) derivation, or proof, within a KBS. For a KBS, execution consists in the construction of the graph, using as "raw material" the given facts and rules which constitute the total KB. Note that the graph does not represent the KB itself.

We are now in a position to define some common terminology, with reference to the digraph model.

2.2.4 Forward-chaining and Backward-chaining

Forward-chaining refers to the temporal order in which the digraph is built. As might be expected, a forward-chaining system (also referred to as *data-driven*) is one which starts with the given facts, and works towards the goal-facts. Typically, the system selects a relevant rule, applies it to the appropriate subset of existing facts (either originally given or already derived), notes the substitutions of particular terms for general terms, puts the newly-derived facts in the graph, tests whether any of them are goals, and if not, repeats the cycle. Clearly, forward-chaining is a constraint, but does not uniquely specify a given order of construction; in working forward, the system has a certain amount of freedom in choosing which of several potential rule-applications it will construct next.

Forward-chaining corresponds to the following intuitive account:

1. We know that Maude's temperature is 102.3° F.
2. A relevant rule is that if a person has a temperature, T, and T is above 100.0, then that person is sick with the flu.
3. Applying this rule to Maude and her temperature, we bind "person" to Maude and "T" to 102.3.
4. As a consequence of this rule-application we derive the fact that Maude is sick with the flu.

A *backward-chaining* system (also called *goal-driven*) is not simply the opposite of a forward-chaining one. A few additional concepts are needed in order to explain the basic idea. A *query* (or *request*) to a backward-chaining system is typically a specification (or pattern, see below) for a goal, for example, sick(person:Maude, disease:?Which), interpreted as "Which disease, if any, does Maude have?". The system tries to derive a fact which satisfies the specification. Note that the query may contain constant terms ("Maude," in this case) as well as variable terms. Also, the rules themselves (not the bindings) may contain constant terms, in their if-parts and then-parts.

Recall that there are three kinds of objects represented in our digraph: 1) facts, 2) the general rules in the rule-applications (one rule per application), and 3) the bindings in

each rule-application, which associate each variable in the rule with some constant term. A constant term may originate from the (direct or indirect) predecessors or successors in the digraph of the rule-application, and the bindings are categorized accordingly. A constant term taken from a query is treated as if coming from a successor.

Bindings whose constants come from successor nodes (either the query or the if-parts of successor rules) are called *request bindings.* Bindings whose constants come from predecessor nodes (either the facts or the then-parts of predecessor rules) are called *assertion bindings.*

Roughly speaking, a backward-chaining system is one which fills in the digraph back-to-front (goal-fact to given facts) for the general rules and for the request bindings and then front-to-back for the facts and the assertion bindings. The intuitive analog here would be:

1. We get a query—What disease does Maude have?
2. We notice that a potentially relevant rule is that people with a temperature above 100.0 are sick with the flu. Possibly, we can use this rule to satisfy the whole query.
3. A specialized form of this rule, then, is that Maude (request binding) is sick with the flu if she has some temperature above 100.0.
4. So, does Maude have such a temperature? A relevant fact is that she has a temperature of 102.3.
5. Returning to the special rule of #3, we note that the temperature 102.3 (assertion binding) is above 100.0.
6. Therefore Maude is sick with the flu.

Referring to Figure 1, a forward-chaining system would fill in the elements in the order A,B,C,D,E. For a backware-chaining system with a query of sick(person:Maude, disease:?Which), the order would be B,C,A,D,E.

Finding a Relevant Rule. Both backward and forward chaining systems face the problem of finding a relevant rule to apply, from within a potentially large rule base and set of facts. At its worst, this might involve checking every rule against every fact (or subset of facts) for each rule-application cycle—a prohibitively expensive procedure. Backward and forward chaining systems both circumvent this cost, but using different strategies.

Because it's got a specific goal to work on, a backward-chaining system is more highly constrained in its search, hence its algorithm can be relatively simple and efficient. It can treat the digraph as a tree (a special case of a graph) with the goal as the root and

given facts as the leaves. Backward-chaining is somewhat more conducive to a global or abstract approach to problem-solving. Because the rules most directly related to the goal are examined first, they impose a kind of grand strategy on the derivation process. Each requirement of the rule-application which is predecessor to the goal can be thought of as a lemma, or "island," in the proof.

Forward chaining systems generally require (and possess) a more sophisticated procedure for identifying relevant rules. One common strategy is to keep track of which rules are applicable to the current set of facts, and then, when facts are added, deleted, or modified, to determine which rules become applicable and which inapplicable [Forg82]. In other words, the entire rule base is not examined every time a change is made to the set of facts; the system can compute directly which *changes* in the applicability of the rules result from *changes* in the set of facts. In contrast to the global problem-solving style of backward-chaining systems, forward-chaining usually has a more local flavor. Since it begins from the given facts, such a system must be able to tell that it is in some sense approaching the goal before the goal is actually reached.

Applicability. How does one decide which strategy is best suited to a particular application? Perhaps the best simple guidance comes from [Brow85]:

> A useful guideline is to have the system start where there is most focus. If the nature of the data is well-understood but the goals are not easily characterized (perhaps because there are many, quite different solutions), a forward-chaining architecture is to be preferred. On the other hand, if there is only one goal and a well-defined subgoal structure but little structure in the input data, it would be better to select a backward-chaining architecture.

2.2.5 Pattern-matching

Pattern matching pervades KBS programming and AI in general. The basic idea is that there is a *pattern*, a kind of abstract description or template for some set of objects, which is compared with some candidate object. The result of the comparison may be a failure—the object failed to conform to the requirements of the pattern—or success, in which case we usually preserve information as to *how* the object matched the pattern, as well as the mere fact that it did.

Pattern matching techniques have been in use for many years, most notably in the SNOBOL language [Gris71], in which the objects being examined are character strings. Typically in KBS programming, the objects of interest are symbols (constants or variables) and structures of symbols, such as the lists and predicates which we've seen above. For example, using the same notation as before, the pattern

[Richard, ?Someone, Marie]

denotes any list with three elements, of which the first is "Richard," the third "Marie," and the second some arbitrary value. If we matched it with the object

[Richard, Paul, Marie]

the match would succeed and ?Someone would be bound to Paul. The pattern

sick(person:Merv, disease:?Which)

denotes any fact about Merv's being sick and which disease he has. If we matched it with

sick(person:Maude, disease:flu)

the match would fail because Maude and Merv cannot be made equal.

Pattern matching can be seen as a generalization of an equality comparison. Note that a constant object can itself be used as a pattern, and when it is, the pattern match reduces to a simple equality test. The usual matching rules, which the examples above used implicitly, are:

1. Two constants match only if they are equal.
2. A constant or structure matches a variable, but the variable is then bound to that constant or structure, i.e., it cannot take on another value within the scope of the match.
3. Two variables match each other, but they are then bound to each other, i.e., they must both ultimately be bound to the same entity, even if they are not yet bound.
4. Two structures match if their corresponding elements match.

The effect of these rules is that a pattern matches an object in just those cases where there exists a consistent assignment of values to the variables in the pattern, such as to make the pattern and object equal. We can imagine "trying out" different values for the pattern variables, until we find a set of values that exactly fits the object. Of course the assignment must be consistent within the pattern, that is, if the same variable appears more than once in the pattern, each occurrence must have the same value.

It is important to distinguish between a variable in a pattern, and a variable of a conventional programming language. While the value of the latter might indeed change during the execution of a program, it is normally thought of as actually having a value at any given point in time. In particular, if the program compares it to some other entity, such

as a constant (e.g., if rate_of_pay = 12.24 then ...), the meaning is simply to check whether the values are equal, not to "try" to make them equal, as does true pattern matching. At the beginning of a pattern match, the variables are truly unbound, i.e., they do not have any value at all, but are available to accept values as dictated by the match.

As might be surmised, one of the main reasons pattern matching is so important to KBS programming is that it provides a very powerful technique for solving the problem mentioned earlier: finding a relevant rule. In the simple case, we can think of the if-part of a rule as containing several patterns, which are then matched against existing facts. A successful match implies that the rule may be applied (i.e., a new rule-application may be created). Recall that the rule-application contains both the general rule, and the bindings. The bindings capture the information as to how the adjacent facts fit the general rule.

There is a further generalization of pattern matching. Not only may a pattern be matched against an object, but two patterns may be matched against each other. Intuitively, the meaning of such a match is to determine whether any object could satisfy both patterns, i.e., whether the intersection of the sets of objects described by each pattern is non-null. If the sets are disjoint, the match fails. If not, the match succeeds and the resulting bindings represent the least restrictive condition for which a qualifying object is guaranteed to match *both* original patterns. For example, if we match:

[Richard, ?Second, ?Third1, Joseph]	(any list of four elements, with "Richard" as the first and "Joseph" as the fourth)

against

[Richard, Stephen, ?Third2, ?Fourth]	(any list of four elements, with "Richard" as the first and "Stephen" as the second),

then the match would succeed and the resulting bindings would be: ?Second bound to "Stephen," ?Third1 bound to ?Third2, and ?Fourth bound to "Joseph". As long as these bindings are in effect, the only lists which would satisfy either pattern are those with four elements, "Richard" as the first, "Stephen" as the second, and "Joseph" as the fourth, as we would expect.

This process of matching one pattern against another is central to the strategy of backward-chaining systems. As the system attempts to proceed backward from a goal pattern, it matches this pattern against those of the *then-parts* of the rules in the KB. The request bindings resulting from a match effectively describe a set of potential solutions, ones which might be generated by the rule, *and* which would satisfy the goal-pattern. This is quite different from the general strategy of forward-chaining, in which facts (not patterns) are matched against the patterns of the *if-parts* of the rules.

2.2.6 Inference Engine

Much of the discussion so far has centered around the expressiveness of the KB, wherein reside the facts and rules of the KBS. Now let us address directly the capabilities of the *inference engine*. The inference engine of a KBS language is the active mechanism which actually builds a derivation (modeled by a digraph), drawing on the facts and rules in the KB. It must implicitly or explicitly keep track of the current (partially-built) state of the derivation digraph, especially including bindings within the rule-applications. It must identify which general rules are currently applicable and must also decide which of these actually to apply next. A language may have a built-in inference engine, or simply provide tools for the user to build one.

2.2.7 Control Strategy

Within the basic KBS model as presented, the essential problem faced by an inference engine is one of ordering. At any point during the development of a derivation, there are typically several places where construction of the derivation digraph may proceed. Which variable should be bound next? Which rule applied? To which set of facts should it be applied? The answers to these questions determine the *control strategy* for the KBS. At a gross level of detail, backward- and forward-chaining themselves represent two basic strategies. What other basic strategies are there? Within a basic strategy, what are the meaningfully different sub-strategies?

Basic strategies are distinguished by which end of the digraph they proceed from. If we picture the work of the inference engine as the construction of a logical bridge between the given facts and a goal, we can grasp intuitively that backward-chaining systems work from the goal back to the facts, and then sweep forward from the facts back to the goal in order to bind variables which have been left unbound. Forward-chaining systems, of course, work from facts to a goal. More sophisticated strategies are possible. Some systems are capable of working from either end, and then meeting in the middle. We shall consider only the two simpler cases.

Within a basic strategy, the question is which of several potential rule-applications along the current leading edge (whether working backward or forward) should be constructed next. Note that the roles of origin and destination are played by the facts and the goal, respectively, for a forward-chaining system, but the reverse holds for a backward-chaining system. Recall that in the digraph a node represents either a fact or rule-application. Typical sub-strategies include the following:

1. **Depth-first:** expand from the node farthest from the origin.

2. **Breadth-first:** expand from the node closest to the origin.

3. **Recent-first:** regardless of distance from the origin within the digraph, expand from the most recently created node.

4. **Best-first:** based on some local evaluation function, expand from the "most promising" node, i.e., the one estimated to be closest to the destination.

5. **Heuristic:** use some domain-specific rule to decide from which node to expand.

Control strategy is important because typically the number of derivable facts is very large, often infinite, and an exhaustive search of all these in order to find a goal is computationally impractical. A control strategy succeeds to the extent that it builds only the nodes which are necessary to connect the facts and goals; insofar as it searches down "dead-end" paths, building superfluous nodes, it is inefficient.

We can better understand the applicability of KBS techniques if we consider the entire spectrum of efficiency. At one extreme, a system would simply explore all paths, regardless of potential usefulness, until it found a goal. This is called blind search. At the other extreme, a system which always could determine exactly what to do next is in effect executing an ordinary algorithm; after all, this is what conventional procedural programs do. It is the middle case, where the system has some, but not complete, information about what to do next—intelligent search—in which KBS techniques are often appropriate and valuable. This characteristic should be one of the main criteria in deciding whether an application is susceptible to KBS techniques.

Information about where to search next is called *control knowledge*. Such knowledge is conceptually distinct from the direct facts and rules of the application domain which we have discussed so far. Roughly speaking, control knowledge is concerned with how to solve problems within a domain, rather than with the facts and relationships of the domain itself. For example, "All fish live in water" and "Birds, and only birds, have feathers" are rules in the domain of zoology. Associated control knowledge might be: "When classifying an animal, first find out whether it has feathers and then, if necessary, whether it lives in water." We have said earlier that rules capture knowledge about how to transform data; control knowledge is about when to transform data. As we shall see in section 3.2, some important issues for KBS languages are whether and how the user can express such control knowledge, and what strategy the language provides in its absence.

2.2.8 Contrast to Conventional Techniques, Revisited

Having surveyed the basic concepts associated with KBS's, let us return to the topic raised in the beginning of this section: what do KBS's do that procedural languages and database systems don't? Abstractly, we've claimed that KBS's offer the modularity and application-independence of databases, together with the power to express procedural knowledge. Now we can see how this works concretely. "Data-like" knowledge can be

represented as facts. Procedural knowledge can be represented as rules. Regarding the representation of control knowledge, however, few generalizations are possible. In conventional procedural languages, of course, this knowledge is encoded with explicit control structures and in the order of statements in the program. In a KBS language, however, controlling the application of the rules may be as much a reponsibility of the system as of the programmer. The execution of a KBS is characterized by the repeated application of various rules in an order determined by the state of the computation, not simply by the structure of the source code.

For instance, if two if-statements are written in succession in a COBOL program, it means that they are to be executed in that order. They will not be re-executed again unless some explicit control structure causes the path of execution to re-enter that body of code. By contrast, two rules written in succession in a KBS language, however much they may resemble their COBOL counterparts, behave much differently. Their syntactic order does not determine when they will be activated. Rather, the rules may be applied (and re-applied) whenever they match the current state of the computation. While users may supply some control knowledge to guide this process, they may leave some or all of the responsibility to the language. Thus, adding a new rule is not as painstaking a process as adding an if-statement to a COBOL program. KBS techniques therefore encourage a more *declarative* style of programming, as opposed to the procedural style appropriate for the expression of deterministic algorithms.

Another way of understanding this distinction is to note that in a KBS, rules, as well as facts, are *data* with respect to the KBS, i.e., passive objects operated on by an active process, namely the inference engine. By contrast, in COBOL, an if-statement is part of the process. It is the ability to treat rules as manipulable objects (even though they capture *procedural* knowledge) that underlies much of the power and flexibility of KBS.

Among the important consequences of treating rules as objects, is that there is a natural means by which the results of a program can be explained to the user. With conventional programming, an answer is presented to a user as a result of "black-box" calculation. An interested user must analyze the source code if he wishes to understand why a particular result was generated. With KBS programming, the derivation digraph, which has already been constructed implicitly or explicitly as a side effect of computation, is raw material for an explanation facility whereby the user can be shown the facts and rules upon which the answer is based.

2.3 Advanced Concepts

The concepts presented so far are fundamental to the understanding of all KBS's. The purpose of this introduction has been to present a very basic common framework within which we may discuss and compare KBS languages. More advanced issues in KBS

design and programming will be covered in the course of the language comparison itself in section 3. There are many sophisticated KBS techniques which this report does not address at all. Among them are:

- How to handle probabilistic data.
- How to handle incomplete or incorrect data.
- So-called higher-order or intensional logic, which is concerned with reasoning about such "indirect" knowledge as the possibility and necessity of facts, or beliefs about facts.

Interested readers should consult the Bibliography, especially [Haye83].

3 Language Support for KBS

In this section, we shall describe in detail which language facilities are available to support the encoding of a KBS system. The discussion shall be organized with respect to the abstract entities of a KBS, such as facts and rules, rather than according to language features.

The treatment of Lisp must differ somewhat from that of Prolog and OPS5. The latter two have a built-in model of a KB and inference engine, and so provide direct linguistic support for representing KBS entities. Lisp does not have such a model built in, although it provides facilities for users to design and implement their own model. Thus, when discussing support for a given KBS feature, we will be comparing the "standard" system-supplied facilities of Prolog and OPS5 to plausible user-defined implementations in Lisp. It should be pointed out that Prolog and OPS5 do not limit users to the system-supplied facilities, but also allow them to design and build customized KBS models.

3.1 The Knowledge Base

The knowledge base (KB) is the passive part of a KBS, where facts and rules are stored. How expressive are the mechanisms provided by the languages? Do they provide users and programmers with a natural way of representing their knowledge? What operations are available for testing the contents of the KB, and for changing them? These are the sorts of issues which will concern us.

3.1.1 Atomic Values

In any KBS, there will be certain values which are taken as primitive. This means that they are not divisible into simpler semantic components and that whatever inherent meaning they are supposed to have can be deduced only within some semantic system which is itself external to the KBS. Atomic values essentially correspond to elementary data types (see Section 4.5, below) and so will not be covered in detail here (for other KB components, the correspondence to language facilities will not be so straightforward). At this point, simply note that all the languages support at least symbols and numbers as elementary values.

3.1.2 Constants and Variables

As explained in section 2.2.2, variables are typically used to represent the general, or unbound, terms in a rule, while constants represent particular values. All three languages have numbers, whose values are, of course, constant. Prolog and OPS5 both implement

variables and symbolic constants, denoted with distinctive syntax. Prolog provides so-called anonymous variables; these are useful as place-holders in a pattern for arguments whose values we do *not* need to capture in the pattern-match. OPS5 uses a null conjunction for the same purpose.

Lisp symbols behave essentially as variables. A special quoted form exists to express symbolic constants. However, the quoted form is typically *not* used to represent constant values in KBS rules and facts. Rather, unquoted symbols are used for both KB constants and variables, and a spelling convention to distinguish the two is adopted and maintained by the programmer [Char80,Wins84a]. Lisp has a feature by which recognition of an character triggers execution of a macro-definition, and this technique is often exploited to implement the spelling convention. Note that such a convention is *not* recognized by the language as such (as is the case with Prolog and OPS5), but only by the program. Prolog and OPS5, then, are at an advantage in that they have linguistically supported symbolic constants.

3.1.3 Facts

A fact in a KB is expressed by a single predicate containing only constant terms, no variables. All the mechanisms provided by the languages for representing facts can be understood as implementations of the abstract predicate-argument model described earlier. This subset of the KB in many ways resembles a relational data base. In both cases, we represent knowledge as a relation among certain entities. The values in a database tuple correspond to the arguments in a fact and the name of the relation to the predicate. Two important differences between a KB and a database are that KB facts need not always adhere to a few fixed formats, and that the argument values needn't be atomic. Table 3 summarizes the way in which facts are represented in the various languages. The major structuring mechanism in Lisp is, of course, the list. Facts may be represented as a list, with the predicate as the first element and the argument values as the rest of the elements. This convention, however, is not in any sense built into the language; it is simply the way Lisp users typically employ the list mechanism to encode facts. The other mechanism provided by Lisp is called a *structure*, which is very similar to a Pascal or COBOL record. It has named fields (called *slots*) and its format must be declared. The type-name of the structure is interpreted as the predicate.

Prolog provides a mechanism which is called either a compound term or a structure, consisting of a functor (which encodes the predicate) and arguments. The arguments are unnamed values, as with the Lisp list, i.e., they are positional, not keyword-style, arguments.

OPS5 has an attribute-value element, consisting of a class name to represent the predicate and attribute-value pairs to represent the argument names (roles) and values,

Table 3: Language Implementation of Facts

Language Feature	Argument Format	Predicate	Argument Name	Argument Value
Lisp list	positional	first list element	n/a	rest of list
Lisp structure	keyword	type name	slot name	slot value
Prolog compound term	positional	functor	n/a	argument
OPS5 attribute-value element	keyword	class name	attribute	value*
Database tuple	keyword	relation name	role	value*

* must be atomic

respectively. As with Lisp's structure, the format of an attribute-value element must be declared. A major limitation, however, is that argument values must generally be atomic, unlike Lisp and Prolog, which allow nesting of structures and lists. The only exception is that OPS5 allows the value of one argument of a predicate to be a list. Thus, OPS5 does not handle recursive data structures very well, although they can be simulated with some effort [Brow85]. Besides these attribute-value elements, OPS5 also provides a vector element, which is simply a list, whose interpretation as a fact is defined implicitly by the program.

Keyword vs. Positional. Thus, we see that all three languages represent facts in much the same spirit, even if the syntax varies. The major distinction is between the declared-format keyword style of the Lisp structure and OPS5 element, and the undeclared, positional style for arguments of the Lisp list and Prolog structure. The former is a somewhat more controlled and documented approach to encoding facts, whereas the latter provides more flexibility and freedom to the programmer.

Of course, either approach may be better, depending on the situation. One important consideration is whether the KB is to contain many facts of a few types (i.e., with the same predicate and roles) or whether it will contain facts of many different types. In the former case, the factual part of the KB (as opposed to the rule part) becomes very much like a data base, and the more controlled keyword approach is indicated. Conversely, in an heterogeneous KB, the freer style may be more appropriate.

An advantage of the keyword approach which always obtains, however, is that it preserves the design information about the predicate. Especially for predicates with many arguments (as often happens in an object-oriented design), keywords make the code easier to read, e.g., "person(name:John, height:70, birthyear:68, weight:166)", as opposed to "person(John, 70, 68, 166)". This advantage applies only to facts, however, not necessarily to rules, in which judiciously named variables can convey the role of each argument.

Constraints. Finally, it is worth noting that all of the languages support the representation and processing of facts only as expressed by definite values for each argument of a true predicate. Let us call these *definite facts*. None provides direct support of derivation based on *indefinite* facts built from boolean operations on definite facts. For instance, assertions such as: "Either John's hair color is red or Suzy's eyes are brown" (disjunction) or "John does not have red hair" (negation) are not easily represented in the factual part of the KB in a way which allows automatic derivation of the logical consequences (in this case, that Suzy's eyes are brown). Nor do the languages provide a way of expressing and enforcing indefinite argument values, such as "John is at least 190 cm. tall", or restrictions on argument types, such as that the value of John's height must be numeric.

All these indefinite assertions may be thought of as *constraints*. A constraint does not imply the truth of any definite fact or set thereof, but does imply the falsity of some. Constraints can be very useful in maintaining the integrity and consistency of a KB. As we shall see, the languages provide considerable support for *testing* whether general conditions such as those mentioned are true for a KB, but not for *asserting* them as facts. See [Fike85] for a discussion on how frame-based representation captures constraint information.

3.1.4 Rules and Patterns

Rules contain patterns in their if-parts, which *test* the contents of the KB, and in their then-parts, to *assert* derived facts. The way in which individual patterns may be combined within a rule contributes to the expressiveness of the language. Also, we should know how powerful and expressive the patterns themselves are. Section 2.2.5 describes the core capabilities of pattern matchers as generalized equality testers and, in particular, their ability to match a structured object, such as a list or predicate. Some languages offer enhanced matching features, beyond this basic model.

Identification of Rules. A rule as a whole is always given a unique name in OPS5. This name may be referred to by the user interface facilities described below (see section 3.3). As we shall see, Prolog allows only a single predicate in the then-part of its rules. The rules are identified by the name of this predicate. Lisp has no intrinsic features to express rules, and so the programmer must define and support any naming conventions.

Computed Expressions in Rules. Section 4 discusses the conventional operations provided by the languages for computing expressions, such as for numeric or character string values. As rules are normally implemented in Lisp, expressions can appear anywhere within a rule. Prolog and OPS5, however, place some restrictions on the placement of expressions. OPS5 allows computation of a value only in the then-part of a rule, and thus cannot easily express a condition such as: "If John is at least two inches taller than Jim...". Although Prolog allows computation only in its if-part and not its then-part, this restriction can be overcome simply by binding the expression's value to a variable in the if-part and then referring to the variable in the then-part.

If-part Features in Lisp. Lisp does not have any built-in pattern matching capability. The user is entirely responsible for the programming of this feature. [Char80] describes a unification algorithm for Lisp, very similar to that provided by Prolog. [Wins84a] presents a somewhat more SNOBOL-like program, in which sequences of elements in the object may be matched by a single pattern element, rather than the usual one-to-one correspondence. The normal trade-off for custom-built software prevails here: the user has more control and can fine-tune the algorithm to his or her specific purpose, but must invest the time needed for development and debugging.

If-part Features in Prolog. Prolog does general purpose pattern matching, including the matching of one pattern against another, as described in Section 2.2.5. In addition to these features, there are also several so-called *evaluable predicates* supplied by the language. These predicates allow matching based on numerical or character comparisons, such as less-than, and greater-than. For instance, the pattern "height(john, X), X > 70" will match if John's height is any number greater than 70. Also, Prolog has a standard total ordering of terms, so as to allow it to compare the order of any two numbers, atomic terms, or predicates. Finally, predicates exist for metalogical testing, such as whether a term is a variable or an integer. These tests are called metalogical because they depend on properties of the program itself, not on those of the domain being modeled.

Prolog has features for the expression of conjunctions, disjunctions, and negations of patterns at the predicate level. Furthermore, these can be nested to an arbitrary depth. Prolog does not directly support boolean operations for individual arguments within a pattern, but these can be easily simulated by first "capturing" the argument value in a variable, and then performing boolean tests on the variable.

If-part Features in OPS5. OPS5 provides much the same capability as Prolog. Patterns can include numerical comparisons and there is one metalogical test to see if an argument is numeric or symbolic. OPS5 does not, however, provide for comparison of non-numeric terms, as does Prolog.

OPS5 supports negation and conjunction at both the predicate and argument level, but disjunction at the argument level only. Thus, in order to express a rule whose antecedent is a disjunction of two separate facts, two separate rules must be written, with the consequent repeated. Unlike Prolog, no nesting of boolean operations is provided. An OPS5 rule must contain at least one positive (non-negated) clause.

If-part Summary. Table 4 summarizes the pattern matching features provided in the if-parts of the rules of Prolog and OPS5. Lisp is not included since its matching facilities depend entirely on the programmer. Overall, Prolog's features provide more power and flexibility than those of OPS5.

Table 4: If-part Pattern Matching

Language Feature	Prolog	OPS5
$=, \neq$	any term	number, symbol, predicate
$<, >, \ldots$	any term	number
Computed expressions	Yes	No
Type test	atom, number, variable	number, symbol
Negation, Conjunction	predicate, argument	predicate, argument
Disjunction	predicate, argument	argument
Nesting of Conditions	Yes	No

Then-part Features. Since the then-part of a rule normally expresses a definite fact or set of facts, rather than a general condition, many of the features found in the if-part do not apply. In particular, the comparison operators and type tests are not used, nor is negation or disjunction. For instance, even though one can *test* whether John's height is greater than 70, one cannot *assert* that as a fact. Recall that the languages do not support constraints.

Of course, variables are allowed within the predicates of the then-part, and these are instantiated as described in section 2.2.4. In Prolog, these variables can be contained

within a data structure, such as a list, and the structure serves as a filter for the application of the rule, since the instantiation of the parameter in question must match the structure. Thus, in Prolog, the then-part may actually do some of the testing normally associated with the if-part.

Because of its backward-chaining strategy, Prolog allows only a single predicate in its then-part. Thus, if a fact has two immediate consequences, Prolog must use two rules to express this, repeating the if-part. OPS5, more conveniently, allows a sequence of actions in its then-part, each of which can generate (or delete or modify, see next section) a fact, thus providing the equivalent of logical conjunction.

3.1.5 KB Operations

So far, we have mostly considered the facilities provided by the languages for testing facts within the KB. In this section we shall discuss how statements in the program can alter the KB during a derivation.

Lisp, of course, does not provide such KB operations directly. It provides some features, however, upon which certain KB operations might be based, under the modest assumption that a KB object will be represented by some kind of Lisp object. Many types of Lisp objects can be created with so-called *make* functions, and modified with the *setf* macro. These normally work on one object at a time. Lisp can also create several objects at once by *loading* them from a designated file. The way in which Lisp might add, delete, or modify KB objects depends on the way in which the KB as a whole is implemented. If a built-in Lisp aggregate, such as a list or hash table, is used then the language provides functions for its manipulation, e.g., for adding, modifying, and deleting elements of the aggregate. User-defined structures would, naturally, require user-defined operations for its manipulation.

There are three ways of manipulating the KB within Prolog. First, the then-part of a Prolog rule consists of the pattern for a single fact. When the rule is satisfied the fact is instantiated, but then immediately discarded. If the same fact is needed later in the derivation or session, it must be proved all over again. See section 3.4.5 for more discussion of persistence of KB operations.

The second kind of operation provided by Prolog is the ability to add or delete individual facts or rules for the entire session, or to delete groups of these. This is accomplished, however, as the side-effect of evaluable predicates which appear in the *if-part* of an applied rule. This is rather counter-intuitive and does not fit well into our general model for derivation, since we normally envision the if-part as testing the contents of the KB, and the then-part as asserting the changes. Since an if-part may contain several patterns, several KB objects may be affected with one rule, even though the then-part, as we've seen, can derive only a single fact.

The third class of operations are those in which a set of KB objects made be read from an external file. These facts and rules may be either added to the KB, or may replace existing objects in the KB with the same predicate name.

OPS5 has statements for adding, modifying, and deleting facts and for adding a rule, but a rule may be deleted only with a user command (see section 3.3). A fact is modified by changing the values of some of its arguments. The OPS5 statements appear in the then-part of its rules, as expected.

Table 5: KB Operations

Operation	Number of KB objects	Type of KB objects	Source	Language Statement	Rule-part containing the Statement
Lisp:					
add	many	fact, rule	file	load	user-defined
add	one	fact, rule	program	make-*x*	user-defined
modify	one	fact, rule	program	setf	user-defined
delete	one	fact, rule	program	remove,	user-defined
		fact, rule		remhash, ...	user-defined
Prolog:					
add	one	fact	program	*implicit**	then
add	one	fact, rule	program	assert	if
delete	one	fact, rule	program	retract	if
delete	many	fact, rule	program	abolish	if
add	many	fact, rule	file	consult	if
replace	many	fact, rule	file	reconsult	if
OPS5:					
add	one	fact	program	make	then
modify	one	fact	program	modify	then
delete	one	fact	program	remove	then
add	one	rule	program	build	then

* does not persist — derived and then discarded

Table 5 summarizes the facilities of the languages for KB manipulation. Prolog's main advantages are its ability to perform the same operations on rules as on facts, and to read in a set of KB objects from an external file. OPS5 has a more natural means of expression in that changes are done in the then-part of rules, and can directly modify facts.

3.2 Derivation

Given a KB, how does a KBS construct a derivation? What capabilities does the language implementation have? What must the user supply? What control knowledge may he contribute to the process and how? These are the major topics covered in this section.

3.2.1 Inference Engine

Both Prolog and OPS5 have a built-in inference engine. Lisp does not. Of course, this leaves the user free to build as specialized and sophisticated an inference engine in Lisp as he or she wishes. If you need flexibility and direct control in this realm, Lisp may be a good choice. Lisp's flexibility is demonstrated by the fact that some implementations of both Prolog and OPS5 are written in Lisp. See [Char80,Wins84a,Forg82] for guidance on the implementation of inference engines. Since Lisp *per se* provides no inferencing, the remainder of this section on derivation will discuss only Prolog and OPS5.

Prolog provides a backward-chaining inference engine. One of the advantages of backward-chaining is that the digraph simplifies to an and/or tree, with the goal as the root. Since Prolog begins at the goal, its derivation is essentially a tree search, instead of the more general and difficult graph search. Prolog performs this search depth-first and left-to-right, where left and right are determined by the order in which the individual patterns are written in the if-part. Thus, the order is static, and cannot depend on the state of computation during execution. While this default strategy is not very sophisticated, its simplicity makes for efficient implementation.

OPS5 is a forward-chaining system. After using the efficient RETE algorithm [Forg82] to determine which rules are potentially applicable, OPS5 then chooses among these, based on one of its two built-in strategies, called LEX (lexical analysis) and MEA (means-ends analysis). For a detailed description of these, see [Forg81]. In general, both strategies favor rule-applications which use more recently generated facts and more specific tests in the if-part of the rule. The motivations are 1) to follow up right away on newly derived facts and 2) the assumption that since more specialized rules apply in fewer cases, they are likely to be the most appropriate for those cases.

Although backward-chaining is built into Prolog, and forward-chaining into OPS5, it is not too difficult for each language to simulate chaining in the opposite direction. OPS5 can do backward-chaining through the use of sub-goals as attribute-value elements; Prolog can do forward-chaining by using a loop control structure. Therefore, while the chaining direction for the application is an important criterion for language selection, it should not be thought of as a decisive factor.

3.2.2 User Control

While the default strategies described above may suffice in the case of simpler problems, they may well need to be augmented to solve larger problems with reasonable efficiency. How do users convey their control knowledge to the KBS?

First, let us deal with a method available within virtually any KBS: users may encode control knowledge simply as more facts and rules. Using KB objects as switches, sub-goals, control flags, and so on, is common practice. Of course, precisely the same sort of thing is done in conventional programs, where items such as end-of-file flags abound. These are control items because they have no meaning for the functional application *per se*, but only for the algorithm. The problem for a KBS, as for a conventional program, is how to make clear to readers of the program which items are for control and which represent real domain knowledge. This responsibility falls directly on the programmer; none of the languages under discussion has any built-in feature to help preserve the distinction between facts or rules which express control knowledge and those which represent pure domain knowledge.

Now let us deal with control mechanisms supported by the languages. As was mentioned, the textual order of clauses (conjuncts or disjuncts) within the if-part of a Prolog rule is quite significant, in that the system will try to satisfy them in that order. Likewise, if a predicate pattern can be matched by two different rules (i.e., the same predicate in two then-parts), the order of the rules in the program dictates the temporal order in which Prolog will try to satisfy them.

Additionally, Prolog has special predicates which give the user some control over the search process. These include one which always succeeds as a goal, one which always fails, one for indefinite repetition (the last two can be used to build a loop), and *cut* which prevents the system from trying to re-satisfy previous patterns in the if-part of the current predicate. See [Cloc84] for a complete discussion of the effects and uses of cut, and the other control predicates. Two other predicates, *setof* and *bagof*, are used to gather up in a list several matching facts for a given pattern. These predicates can be used to cause Prolog to search a sub-tree in breadth-first order, rather than depth-first.

In OPS5, the order of clauses in the if-part of rules, and the order of the rules themselves, does not affect control strategy, as happens in Prolog. The then-part, however, consists of a series of actions which are executed in order, and this order may be significant. In particular, if several facts are created by a then-part, their relative recency will be determined by the sequence of actions creating them, and this may affect which rule-application is constructed next. Recall that OPS5 attempts to process more recently created facts first.

The only feature provided by OPS5 for the express purpose of affecting control is the *strategy* command, which designates either LEX or MEA as the strategy to be used.

In summary, Prolog allows the user a bit more control over the derivation process than does OPS5, at the price of being somewhat more order-dependent, hence forcing the user to pay attention to control issues whether he wants to or not.

3.2.3 Goals and Termination Conditions

As a backward-chaining system, Prolog has a very natural model for specifying a goal. The user enters a goal-pattern for which Prolog attempts to derive a matching fact. If it succeeds, it reports the success, together with the bindings of the variables in the goal-pattern with which the derivation completed. If it explores the entire goal-tree without finding a derivable instantiation of the pattern, Prolog reports failure. After a success, the user can either enter a new pattern, or request that other solutions for the same pattern be reported. If the latter, then Prolog resumes its tree search where it left off, and finds any other derivable fact which matches the pattern.

The goal-pattern functions exactly like the if-part of a rule. It is useful to think of the user query (the goal-pattern), as a rule of the form: "If query then success" temporarily added to the KB. The entire derivation process then becomes an attempt to construct a valid rule-application for this rule.

In OPS5, the system simply keeps applying rules as long as any are applicable, or until the action *halt* is executed in the then-part of a rule. OPS5 will not apply the same rule to the same set of facts, and so will not loop indefinitely unless it can generate something in a new way. Unlike Prolog, it is not assumed that the last fact generated is the goal, or all of it. Intermediate facts may also constitute part of the answer; this is under the control of the user.

3.3 User Interface

Our main concerns in this section are the types of built-in commands and queries by which a user may interact with an existing KBS system. Aside from built-in features, I/O facilities may, of course, be used to build a customized user interface. Typically, the I/O statement accepts or displays argument values during pattern-matching. For information about the I/O facilities, see section 4.6.4.

This section will cover all the relevant run-time commands and statements provided *directly* by the language implementation. Of course, it is always possible to build more elaborate interfaces than those directly supplied. We shall not attempt to distinguish between KBS-oriented and general-purpose features, nor between those used for debugging and for normal execution. A *command* may be issued directly by the user as he interacts with the system. A *statement* is a syntactic entity within the language, written as part of the program. Some commands are also statements; these shall be noted as they come up.

In both Lisp and Prolog, there is a debugger separate from the regular system. Prolog defines a distinct set of commands for each, although many commands are accepted by either. Lisp defines only functions implemented by the regular system. The standard says that a debugger should exist, but its nature is implementation-defined. In OPS5, there is conceptually only a single system which accepts all commands, whether or not for debugging.

Some cautionary notes: first, the languages all use very different vocabularies to describe their user interface facilities. Terms such as "break" and "trace" are *not* applied consistently across the languages. Our description will attempt to use terminology according to the most commonly accepted meaning, but this might not correspond to a language's own documentation. The second point is that the user interface is probably one of the most variable features of a language across implementations. What follows is a conservative estimate; many implementations offer a richer set of facilities. Finally, there is not always a one-to-one correspondence between a given capability and a command. This section describes the capabilities built in to each language; in some cases, one may invoke the feature in question with two commands, or different features may be invoked with different options of the same command.

Table 6 provides a highly abbreviated summary of the user interface facilities of the languages. It is not self-explanatory; please refer to the accompanying documentation and the language references themselves [Forg81,Pere84,Stee84] for the full meaning of the entries.

3.3.1 Invoking and Exiting

Invoking. Not too surprisingly, all the languages provide a way for users to enter the system. In Lisp and Prolog, one starts a derivation simply by entering the name of a top-level predicate, together with any arguments. This is defined by the program, not the language. "Top-level" simply means that the routine is set up to be called directly by the user, as opposed to utility routines which may be used by other parts of the program, but are not suitable for direct user interaction. See section 3.2.3 above for more detail on Prolog. In OPS5, one always starts up the system with a simple *run* command.

Exiting. In Prolog, one can exit a derivation, but stay within Prolog, or exit Prolog altogether. OPS5 provides no way to exit from the derivation without also exiting from OPS5 itself. There is no standard exit convention for Lisp.

Table 6: User Interface Features

Language Feature	Lisp	Prolog	OPS5
Top-level control:			
invoke derivation	Yes	Yes	Yes
exit derivation		Yes	
exit system	undefined	Yes	Yes
Watch derivation:			
complete		Yes	Yes
selective	Yes	Yes	
Pause and step:			
pause from program	Yes	Yes	Yes
pause at named entity		Fact or rule	Rule
pause after n cycles			Yes
asynchronous interrupt		Yes	
step thru named entity	Yes, via statement	Yes	
step thru all	Yes	Yes	Yes, via *run*
Inspect KBS:			
named KB object	Yes	Yes	Yes
matching KB object		Yes	Fact only
derivation-state		Goal stack	Conflict set
Manipulate KB:			
add KB object		Yes	Fact only
delete KB object		Yes	Yes
Manipulate derivation:			
abort		Yes	
backup		Yes	Yes
continue	Yes	Yes	Yes
suspend derivation		Yes	

3.3.2 Watching the Derivation

Suppose the user wants to watch the progress of a derivation as it is being constructed, without pausing. Prolog provides a way to watch either the entire derivation (complete trace), or just the parts of it having to do with certain predicates (selective trace). In Lisp the user can designate selected functions to be traced. How such a trace corresponds to the derivation process depends on how the user has implemented the inference engine. Finally, OPS5 supports a complete trace, but not selective.

3.3.3 Pausing and Stepping

We will adopt the term *pause* to mean stopping the progress of a derivation at some intermediate stage so that the user can examine and modify the state of the computation. Lisp and OPS5, but not Prolog, refer to this as a *breakpoint*. *Stepping* means essentially pausing after every atomic action, so that the user can watch or intercede in the derivation one step at a time. Typically, the user can advance computation each step with a single keystroke.

Pausing. All three languages provide statements within the program which will cause a pause when encountered. Prolog has a command which specifies a given predicate (acting as either a fact or a rule) as a pause point. Also, Prolog, uniquely among the languages, defines an asychronous pause (interrupt). OPS5 can specify a rule, but not a fact, as a pause point. OPS5 also allows a numeric argument on its *run* command to specify the number of match cycles the system should go through before pausing for further user interaction. Thus, at the next pause, OPS5 will have built at most that many additional rule-application nodes in the derivation digraph.

Stepping. Both Lisp and Prolog allow stepping through a derivation. OPS5 does not support single-keystroke stepping, but can achieve the same logical effect by repeating the *run* command and designating the number of cycles as 1. Prolog has a command to designate a certain predicate as the place to start stepping, when encountered within a derivation. Lisp can do this also, but using statements within the program, not commands.

3.3.4 Inspecting the KBS

The languages provide many facilities with which the user can inspect parts of the KBS, typically during a pause. These facilities do not affect the logical behavior of the KBS.

Inspecting KB objects by Name. Lisp provides an *inspect* command which allows the user to view interactively any named data structure. Prolog has predicates to list rules or facts according to the name of their identifying functor, either all at once or one at a time. In OPS5, it is possible to display any of the rules, referring to each by its unique name. Also, one may display a fact, based on the fact's unique time tag (an internal numeric identifier).

Inspecting KB objects by Matching. At a pause in Prolog, the user may suspend the current derivation and then issue a Prolog query in order to display any matching fact or rule in the KB. OPS5 has two specific commands, one to display a fact matching an arbitrary pattern, the other to display facts partially matching the if-part of a specified rule.

Inspecting the State of the Derivation. Prolog, since it does a tree search, has a command to display the current goal stack of the derivation tree, i.e., everything on the path between the point reached so far and the original goal. In OPS5, one may inspect the *conflict set* during a derivation. This is the set of rules which are currently applicable, together with the associated enabling facts for each rule. It is from among this set that the strategy will choose a rule actually to be applied next.

3.3.5 Manipulating the KBS

We will now discuss commands with which the user can affect the logical behavior of a derivation, either by altering the KB, or by re-directing the derivation process.

Manipulating KB objects. Both Lisp functions and Prolog predicates may be issued either as commands or as statements. Therefore precisely the same KB operations performed by the statements described in section 3.1.5 are available as user commands. OPS5 has commands for adding or deleting facts, and for deleting a rule, but not for modifying a fact or adding a rule.

Manipulating the State of the Derivation. Prolog has a large set of commands for directing the derivation process. A user can abort the derivation, retry the current subgoal, continue by stepping, continue without tracing, continue after the current subgoal, or suspend the current derivation and issue any normal Prolog command. OPS5 has a command to back up a given number of rule-application cycles, or the run command may be used simply to continue. Lisp guarantees only that there will be an implementation-defined way to continue.

3.3.6 Summary of User Interface

Considering the user interface as a whole, Prolog provides the most extensive set of features, Lisp the least. Of course, Lisp's facilities can be only general-purpose, not KBS-oriented, since Lisp itself has no built-in model of a KB or of derivation. The base document for OPS5 is notably older than for the other two languages, and so it may represent a somewhat conservative picture of the OPS5 environment. For instance, even though [Forg81] does not define a stepper for OPS5, many implementations may provide one. Again, in general, it is worth emphasizing that an actual language processor may have a far richer repertoire of commands than those specified in the base documents.

Leaving aside debugging facilities, note that Prolog and OPS5 differ strongly in their models of how the user will normally interact with the KBS. Broadly speaking, the user can inform the system what he wants in two ways: 1) by specifying a goal-pattern (i.e., find me something like this), or 2) by telling the KB some new user-specific facts, and seeing what it derives as a result. In other words the user can approach the derivation from the side of the goal facts or the given facts. Prolog, being a backward chaining system, is oriented towards the first method, but can utilize the second to a great degree as well, since it has features for direct manipulation of the KB. OPS5 is limited to the second approach, unless the programmer explicitly builds in a goal structure visible to the user.

3.4 Dimensions of Knowledge Design

The purpose of this section is to discuss some of the more practical aspects of the implementation of a KBS. So far, we have presented the KBS rather abstractly, as something which constructs a derivation, working from a KB. Where does the KB come from? Who develops it, who maintains it? Does the system provide any built-in facts or rules? Is the KB just a set of facts and rules, or are there other levels of organization? Language and design issues are intertwined in these areas, and we shall discuss both.

Some of the following sections omit a discussion of Lisp. In these cases, simply assume that the language does not encourage any particular approach.

3.4.1 Review of Knowledge Types

Broadly speaking we have seen three kinds of knowledge to be incorporated in a KBS:

1. **Facts,** which capture specific information about some entity in the domain of interest. Corresponds closely to the intuitive notion of data, such as is recorded in a database.

2. **Rules**, which encode general procedural knowledge about which facts may be inferred from known facts. Viewed more declaratively, one may think of rules as representing known relationships among facts in the domain.

3. **Strategy**, (control knowledge) which is used by the system to decide when a given rule should be applied.

The first two of these constitute the KB of the KBS. The third is either implicitly incorporated in, or used by, the inference engine as it builds a derivation digraph. This categorization is somewhat blurred in that facts and rules may embody control knowledge (see section 3.2.2, above) as well as pure domain knowledge.

For each of these three types of knowledge, one may ask the sorts of questions mentioned above. The answers for a given KBS provide a characterization of it within the space of possible implementations.

While all three languages provide some means for expressing knowledge of all three types, it should be noted that OPS5 is particularly rule-oriented. At the beginning of a derivation, the OPS5 KB contains only rules and no facts. (The OPS5 KB may also contain the declarations of the format of facts, as described in section 3.1.3, but not the facts themselves). All facts are generated during execution, even if by a special initialization rule whose only purpose is to generate them. Since a permanent KB may well need to contain facts as well as rules, this is a somewhat awkward constraint.

3.4.2 Declarative vs. Procedural Style

In declarative programming, the order of statements is insignificant. Therefore, the development and maintenance of code becomes easier, since the programmer needn't be concerned with the location of the code being manipulated. Conversely, the meaning of procedural code is strongly dependent on the sequencing of statements, and is accordingly more difficult to manage. Thus, it is important to know whether a language supports a declarative or procedural style for KB objects. Note that just because strategy is about the order of inference, it does not follow that it must be expressed procedurally; one could *declare*, for instance, that a search is to be done breadth-first.

Whether the representation of knowledge in Lisp is declarative or procedural is, of course, a function of the program, not the language. It is common to use lists as the means to express collections of facts and rules. Since lists are searched sequentially, the order of KB objects in a list is likely to be significant at least for performance, and perhaps logically as well. Other implementation strategies (e.g., representing the KB with user-defined structures, such as networks or trees) will, of course, have their own characteristic effects.

In Prolog, control knowledge is expressed procedurally. Both the order of KB objects for the same predicate, and the order of clauses in the if-part of a rule are significant. As we've mentioned, this lexical order corresponds to the left-to-right ordering within Prolog's tree search. This has consequences, not only for performance, but also on the logical behavior of the program. For instance, two rules which work in one order may generate an infinite loop if their order is reversed.

OPS5 is the most declarative of the three languages. The lexical order of KB objects has no logical effect, and strategy is declared with a single command or statement. The order of statements in the then-part of a rule, however, may have some effect. These statements are executed sequentially and the resulting state of the facts in the KB, for instance, can depend on which modifications are done first. Also, as was mentioned earlier, OPS5's strategy takes into account the recency of facts, which is in turn determined by order within the then-part.

3.4.3 Depth and Width of KBS

KBS applications may be categorized according to the degree to which they depend on large numbers of given facts and rules, vs. depending on a great deal of inference to achieve their results.

Width. Wide applications use large numbers of given facts and rules. Typically, the situation being modeled has many factors relevant to the outcome, and the rules tend to be ad hoc and based on experience, rather than on some elegant theoretical foundation. An example might be diagnosis of automobile problems, in which the system can call on a large knowledge base encoding information about different makes of cars, problems commonly found with certain kinds of cars, relationships between causes and symptoms, and so on. By contrast, a narrow application relies on a few facts to capture the relevant information. In a puzzle-solving system, for instance, the state of the puzzle is the only relevant factor, and it is usually expressed in one relatively small structure.

Depth. In deep applications, the distance between the goal and the given facts in the derivation digraph is large. Typically, the same rules are re-applied many times to different facts, in order to arrive at the result. Again, puzzle-solving systems, in which a great many moves may have to be made, according to only a few rules, are a good example. In shallow applications, there may be only a few simple inference steps from the facts to the conclusion. The power of an automobile diagnosis system probably derives not from its powerful reasoning capabilities, but from the large set of facts at its disposal.

Recalling the earlier discussion on KBS vs. conventional programming (see section 2.2.8), we may say that wide applications correspond more to the "database" aspect

of KBS, while deep applications most fully exploit the capabilities associated with procedural knowledge.

While any of the languages can handle either deep or wide applications, Prolog is probably more attuned to depth and OPS5 to width, since the OPS5 implementation keeps track of the applicability of all rules to all facts, but forces the user to express all inferences explicitly, while Prolog automatically builds deep inference chains, but performs a linear search on facts.

3.4.4 Global Organization of KB

Prolog gives the user a very unified view of the KB. Facts and queries are treated as special cases of rules: facts can be thought of as rules of the form "If true then fact" and queries as "If query then success", where "true" and "success" are built-in values. Thus, within the KB there is no inherent distinction between facts and rules; the KB is simply a sequence of KB objects. Prolog also offers a keyed database facility, for storing and accessing KB objects according to the value of the key. For applications which rely on a large set of facts for which one of the arguments is a natural identifier (such as social security number for people), this might be useful.

OPS5 strongly distinguishes facts, which are called *working memory elements* and reside in working memory, from rules, which are called *productions*, and reside in production memory. Aside from this bifurcation of the KB, however, no further internal structure is provided.

Neither Prolog nor OPS5 has any built-in feature to support modularization of the KB objects into logically related groups. Any structure internal to the set of facts or rules must be built and maintained by the programmer. Prolog does at least define a mechanism to read a file of KB objects into the KB, encouraging users to segment a large program into smaller files of related objects. Lisp's package facility (see section 4.4) provides a good basis for logical modularization when implementing a KB.

3.4.5 Persistence of KB Operations and Monotonicity

Operations on objects in a KB, including additions, modifications, and deletions, may permanently alter the KB, or have only a transient effect. In the list below, we distinguish among a few typical timescales for such changes.

1. **Permanent.** The change permanently alters the KB, which itself is stored on some persistent medium. This is analogous to an update on a file or database, which is effective until explicitly reversed by a later update.

2. **Session.** The change is made during a user's session with a KBS, and stays in effect until the user exits from the system. At the start of the next session, however, the user will see the KB in the same state as it was at the start of the previous session.

3. **Momentary.** A KB object is discarded as soon as it is generated.

Prolog allows the user to write out KB objects to a file, and then read them back at a later time. Likewise, OPS5 provides for file I/O of KB objects. Lisp supports I/O on any kind of Lisp object, and so would be capable of doing KB file access for any reasonable representation of a KB. So while none of the languages has direct support for permanent updating of a KB, they provide simple file processing features needed to construct and read a KB. Non-permanent changes to the KB are discussed in section 3.1.5. As noted there, the effect of most operations persists for the session. The conspicuous exception is the predicate in Prolog's then-part, whose instantiation as a fact is momentary.

Note the differing stability normally associated with each of the three components of a KBS. Typically control knowledge is comparatively stable. The user may change strategy a few times in a session, but far more often, the KBS always uses the same strategy.

As for rules, we've seen that the languages do provide facilities for adding or deleting rules during a session. Nonetheless, it is more typical for a KBS to work with an essentially static set of rules. Normally the rules are changed only in the course of maintenance of the KBS, either to fix a bug or to incorporate additional knowledge in the KB.

Facts are typically the most volatile component of a KBS. They may be entered or changed by the user, to describe the special aspects of his problem, or they may be generated in the course of a derivation.

This is a good place to point out that the explicit addition or deletion of facts may violate the abstract model of *monotonic* reasoning that we've implicitly adopted so far. For our purposes, the idea of monotonic reasoning is that all facts or absences of facts are permanent. The derivation is conceived as a static, timeless entity, without regard to the order in which it was built. If one rule-application within the derivation digraph depends on a given fact, then presumably any other rule-application should be able to depend on that same fact; otherwise, we have a situation in which a fact is true in one part of the derivation and false in another.

But this very possibility is raised as soon as the user is allowed arbitrarily to add and delete facts (adding is just as potentially inconsistent as deleting, since an earlier rule may have depended on the *negation* of a fact). For instance, suppose we have a line of reasoning such as "If Bob is taller than Jim, and Jim is taller than Joe, then Bob is taller than Joe", and add "Bob is taller than Joe" to the KB. If, at a later time, we delete "Bob is taller than Jim" it is now unclear whether the original conclusion is still supported.

Note that Prolog's inference engine, which generates implicit facts during backward chaining, is designed *not* to violate monotonicity. Only explicit program statements to alter the KB can cause this. Thus, one can write a Prolog program using only its implicit derivation and thereby avoid any potential problem. But in OPS5, the only way to affect the KB is with explicit statements, and so there can be no automatic protection from inconsistency.

3.4.6 Domain Dependence

The types of knowledge in a KBS differ with respect to their typical dependence on the problem domain. The facts are usually almost entirely domain-dependent. They represent either direct truths of the domain, or domain-dependent control knowledge. While most rules capture domain knowledge, a KBS might well draw upon some set of certain common-sense rules which would be useful across many applications. Rules to encode mathematical knowledge, for instance, might be appended to the KB's of various applications. Prolog supplies so-called *evaluable predicates*, which are simply a set of general-purpose rules. OPS5 embeds domain-independent knowledge in functions, rather than as part of the KB.

In theory, strategy may be almost completely domain-independent. This surprising capability is one of the hallmarks of KBS's. Realistically, however, only relatively simple applications can afford to rely entirely on a system's default strategy, whether it be Prolog's backward-chaining or OPS5's forward-chaining. Larger applications must give some information to the system regarding domain-specific problem-solving techniques.

3.4.7 Roles and Responsibilities

Let us ask who has responsibility for generating the three types of knowledge in a KBS. We shall distinguish four roles assumed by those who interact in some way with the KBS. Of course one person may take on several roles.

1. The **end-user** of a KBS wishes to take advantage of it in order to solve a problem more easily than he could alone. He does not contribute to the expertise of the system, but rather conveys to the system the knowledge about his particular situation, so that it can bring its expertise to bear.

2. The **domain expert** helps to develop the KBS by contributing his knowledge to be encoded. This may well include domain-specific control knowledge (how to solve problems) as well as the direct facts and rules of the domain.

3. The **knowledge engineer** (also known as the programmer) helps to develop the KBS by contributing her understanding of KBS techniques and practices. She and the domain expert collaborate closely in building, testing and maintaining the KBS.

4. The **system** is the set of software tools which will be used by the knowledge engineer to build the KBS. We have been discussing language systems, but of course this role may be filled by other sorts of packages, such as expert system shells.

Note that throughout this report, when we refer to the "user" of a KBS, we mean someone filling any of the first three roles. The term "end-user" is reserved for the specific meaning described above.

The facts in a KB may originate with someone in any of the four roles. The end-user will tell the system about his circumstances. These end-user facts will probably persist only for the session and not form a permanent part of the KB. The domain expert will contribute the enduring factual knowledge to be captured by the KB. The knowledge engineer is least likely to generate facts, but may well help the domain expert design the representation of the facts. For instance, she might realize that for this application, a property-oriented representation is more felicitous than an object-oriented approach. Either the knowledge engineer or the system may contribute some very general-purpose, domain-independent facts.

Rules follow somewhat the pattern described for facts. The content of the rules will generally come from the domain expert, who may consult with the knowledge engineer as to the design of their representation. The knowledge engineer or system may provide general-purpose rules. The end-user, however, is far less likely to provide rules rather than facts, since knowledge specific to the end-user is normally expressed by atomic values rather than general relationships. In the OPS5 model this dichotomy is assumed to be rather sharp. The permanent KB consists only of relatively stable rules (presumably formulated by the expert), while the end-user is expected to contribute only facts. Conversely, Prolog makes little distinction between facts and rules, and allows the end-user access to either.

The end-user will almost never have access to the control knowledge of a KBS. It is for this component that the expression of domain knowledge by means of KBS techniques is likely to be most complex; accordingly the domain expert and knowledge engineer must collaborate very closely if the system is to have a strategy which exploits domain-specific problem solving techniques. Of course, the system may well provide a default strategy, as we've seen for Prolog and OPS5. A major design issue for most KBS builders is whether this system-supplied strategy will be adequate or whether they will have to design in control knowledge explicitly oriented to the domain.

3.5 Flexibility and Power for KBS Applications

A KBS language may provide the user a large and flexible set of mechanisms for detailed representation and control of a KBS, or a few very powerful features which constrain the KBS model, but require less detailed programming. The first attribute, flexibility, is appropriate for a highly skilled, professional KBS architect. The second, power, is most needed by comparatively casual users. These attributes are not necessarily opposed, but in practice a language will usually emphasize one or the other.

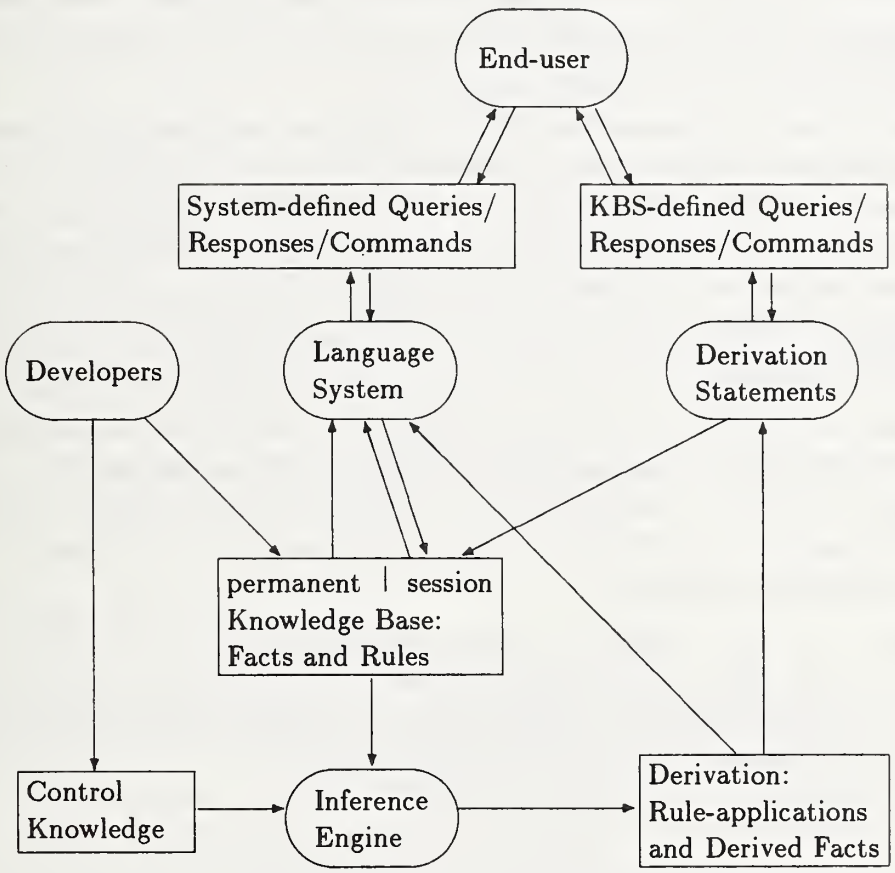
This is the criterion which most strongly differentiates the three languages under study. Lisp provides a great deal of flexibility and fine control. As we shall see in section 4, Lisp, as defined in [Stee84], has a great variety of conventional structures for expressing data and control. As has been mentioned, however, Lisp does not directly support the representation of a KB, nor does it provide an inference engine, as do the other languages.

Prolog and OPS5, on the other hand, give the user a ready-made model for derivation and knowledge representation around which the application must be designed. Perhaps it is fair to say that OPS5 emphasizes power even a bit more than Prolog, since its features for both knowledge representation and controlling strategy are more constrained than Prolog's, and its default strategy more sophisticated.

3.6 Summary of Language Support for KBS

Now that we have surveyed KBS issues both in the abstract, and as supported by three KBS languages, let us review the main points of importance. In conventional programming, data is used to represent specific facts within a domain and an active procedure encodes both general, logical relationships within the domain, and the control for applying these general rules to the facts. A KBS system provides an at least partially automatic control process for applying rules. The user need only enter facts and rules in a declarative style. The system also provides a rudimentary user interface to the derivation process, including KB queries and responses. Figure 2 presents a simple model for the flow of data within a KBS. Not all possible relationships are shown, just those most typically found. Rectangles represent data structures, and ovals, active processes.

Figure 2: Typical Data Flow in KBS



4 Conventional Language Factors

In this section, we shall examine language features which are not peculiar to KBS techniques. Even though KBS capabilities are not at stake, it is still desirable that a language have a rich enough set of features that it can support a variety of applications. For instance, if an expert system application requires the use of complex numbers, it is clearly advantageous for the language to support this data type.

As a generalization, one can safely say that Lisp directly supports a far greater variety of conventional features than does Prolog or OPS5. Lisp, of course, has been evolving since the 1950's, and has acquired features in response to the many demands made on it in that time. By contrast, Prolog and OPS5 have graduated from the academic environment comparatively recently, and have not taken on all the apparatus associated with programming languages in commercial use.

The lack of direct support for some feature does not, of course, preclude the possibility of implementation of that feature by the user. For instance, although Lisp supports rational numbers and Prolog does not, it is not an especially daunting task to build a simple model of rational arithmetic in Prolog. The question is the degree of effort required. Generally, we shall not attempt to estimate that effort, but users should be alert to the possibility (and, sometimes, the difficulty) of programming a needed feature in an otherwise satisfactory language.

This report is primarily concerned with KBS applications, and so we will not attempt to cover every detail of the unrelated features of the languages. Since Lisp has by far the largest set of these features, such an approach would, in effect, amount to an introduction to Common Lisp. Readers who are interested in a full exposition of Lisp capabilities should refer to [Stee84].

This section will cover intrinsic language features in the earlier subsections, and extrinsic features in the later. The intrinsic features are those stemming from the logical definition of the language, while the extrinsic ones are those which have to do with the way in which the language has actually been implemented.

4.1 Syntactic Style

The three languages do not have differences in style which offer any major advantage or disadvantage. All are free format, all allow long identifiers, and none requires the declaration of variables. In other words, all are oriented towards a dynamic style of programming, in which entities are freely created and destroyed. OPS5 follows Lisp conventions quite closely, in that all entities are expressed either as a symbol, or as a parenthesized list of symbols.

4.2 Control of Execution

In this section, we address those language features which are used to describe algorithms. In conventional languages, we normally have a sequential, single-thread model of control flow. This model does not always explain fully the procedural behavior of KBS programs. Let us ask to what extent each of the languages is described by a sequential model.

Lisp is essentially based on the sequential model, although with an applicative style. That is, much of Lisp's semantic behavior is expressed with function invocations, rather than with a set of sequentially executed statements. Nonetheless, within a function, statements are executed in the (static) order in which they appear in the program.

Prolog, like Lisp, leans heavily on the notion of invocation, although in Prolog's case, one invokes a predicate to be instantiated, rather than a function to be evaluated (see section 4.2.3). But again, execution proceeds as dictated by the order of clauses within a rule; recall that this corresponds to the left-to-right ordering of searching within Prolog's goal tree. So, although one may understand Prolog programs in terms of an abstract declarative semantics, as a practical matter, programmers often must take account of the order in which clauses are to be satisfied. See section 3.4.2 for more information.

Finally, in OPS5, the sequential model becomes completely inapplicable, in that the temporal order of rule-application is unrelated to the lexical order in which rules appear in the program. Thus, in OPS5, the programmer does not, in fact, exert much control over execution; this is taken over by the OPS5 inference engine and default strategy.

4.2.1 Structured Programming

We use the term "structured programming" in the narrow sense to mean those language constructs which determine sequence of execution at the detailed level. Lisp has a rich set of such features, including those for two-way or multi-way selection (if-then-else and case), and for various kinds of iteration. Prolog has only an if-then-else construct, and OPS5 has no features for traditional structured programming.

4.2.2 Blocks

Blocks allow the programmer to mark off sections of code to be treated as a single statement. Within the block, data may be defined locally and is inaccessible from outside the block. Only Lisp supports blocks.

4.2.3 User-defined Functions and Subroutines

Lisp and Prolog allow the user to write a module of code which can be invoked from anywhere in the program or invoked recursively, and which serves as the normal scope of data.

In Lisp, functions are the main structuring mechanism (but also, see section 4.4, below). Although, in the abstract, the only effect of a function is to return a value based on its arguments, in fact Lisp functions are often executed for their side-effects, i.e., as if they were subroutines.

The invocation of a Prolog predicate does not correspond exactly to the traditional model for either functions or subroutines. From one perspective, invoking a predicate is much like calling a boolean function: the result "returned" is effectively either true (success) or false (failure). Prolog does not allow the user to define functions to return non-boolean values, e.g. numeric. Unlike a pure boolean function, however, the arguments passed to the predicate may not all have values at the time of invocation. Unbound variables are "filled in" by (successful) Prolog pattern-matching, somewhat similar to the way a conventional subroutine may return values through parameters. Furthermore, it is possible to write Prolog predicates so that different combinations of parameters can be input to or output from the predicate on different invocations. That is, unlike traditional subroutines, where each parameter is statically designated as input or output, a given Prolog parameter can be either a constant (in which case it acts as input), or a variable (output), or a structure containing both.

OPS5 does not support the definition of conventional functions or subroutines within the language itself, but has an escape mechanism whereby the user can define these within the implementation language of the system, which may be either Bliss or Lisp. Note that OPS5 does not have recursive procedures, although, as with recursively-defined data structures, these can be simulated [Brow85].

4.3 Control of Data

This section discusses the mechanisms available to the programmer for specifying the extrinsic characteristics of data, such as its scope and persistence. The actual data values and operations are covered in section 4.5.

4.3.1 Scope of Data

In Prolog and OPS5, the scope of data items is the major structuring mechanism for each language: for Prolog, the predicate, and for OPS5, the rule or working memory element. For Lisp, however, the scope may be either the function, for parameters, or global,

for non-parameter data. Prolog and OPS5 provide no global data at all (other than the facts in a KB). See section 3.4.4 for remarks on the global organization of the KB. Lisp's *package* system, described below, allows programmers to set up a series of name spaces, so that data items which might accidentally be given the same name can be distinguished, based on a unique prefix.

Thus, Prolog and OPS5 rely on their control modularization to prevent unintentional references to data. Their module size tends to be rather small, all data is passed by parameter or through pattern-matching, and global data is not supported. As a result, we get a very "local" and simple model of data access. Lisp, on the other hand, has a series of features for the programmer to build more elaborate data visibility into the program, but also is more subject to unintended effects.

4.3.2 Data Type Checking

In Prolog and OPS5, a variable cannot be declared to be of a given elementary type; any variable can take on any legal value. An error may result, however, if the program attempts to perform an operation on an unsuitable value, e.g., multiplying a non-numeric value. Lisp works the same way in default mode. However, it also allows type declarations for variables and functions. Lisp also provides a limited set of coercions, whereby a value of one type may be converted to another type. See section 3.1.3 for a discussion of the typing of aggregate data.

4.3.3 Data Abstraction

Lisp provides for the definition of new data types, based on those given by the language. Existing types can be combined, extended and restricted in various ways to form the new types. The user-defined types can appear in declarations, and be used freely throughout the program. Prolog and OPS5 have no such facility. Of course, in OPS5, one can declare the format of a working memory element to represent facts, (see section 3.1.3) but no other features for type construction are provided.

4.4 Packages

A package system has many features to help the programmer in the construction of a large system. Packages represent a more abstract level of structure than that found separately for data or control. In a package, logically related data and operations can be defined together, and the entire package invoked as needed. Lisp is the only KBS language which supports this structuring technique (among conventional languages, there is a similar facility in Ada). Lisp packages can be grouped together into *modules,* which can

then depend on each other in various sophisticated ways. In particular, certain modules can declare their dependence on other modules, and the system will see to it that the required modules are incorporated into the final system. As mentioned above, the default scope of data is the package, so name conflicts between packages don't arise.

4.5 Elementary Data Types and Operations

We will now look at the actual types of data, and operations thereon, directly provided by the languages. Elementary data types (also called scalar) are those which represent a single indivisible value, as opposed to aggregate types (covered below) which represent some combination of values. For traditional programming languages, these features are typically where much of their semantic power is to be found. This is largely true of Lisp, but not so for Prolog and OPS5.

4.5.1 Symbols

The support for symbols as a data type distinguishes all three languages under study from conventional languages. The reason that symbols are a central feature of KBS languages is that these languages need to represent and manipulate concepts and objects of arbitrary kinds, just as other languages deal with certain restricted kinds of objects, such as numeric and character, as part of their domain. Symbols, then, are the linguistic means for representing objects or concepts, just as numerals stand for numbers and character strings represent text. And, just as linguistic operations on numerals and character strings model natural operations on the represented entities, so symbolic *predication* models relationships among the represented objects and concepts.

A symbolic value typically corresponds to a natural language word or phrase. Thus a symbol is normally self-explanatory, just as is a numeral or character string. Symbols, however, should be distinguished from character strings, which are aggregates of characters. Perhaps the closest analog in traditional languages is the enumerated type, in which certain identifiers are taken as the primitive values of the type, e.g. *red, orange,* and *yellow*, as values of type *color*.

There are only a few operations to be performed with symbols: one can combine them into larger data structures (such as lists or predicates), copy them, or compare them for equality. All the languages do these things. In addition, Lisp and Prolog treat symbols as having alphabetic order, allowing them to be compared with a less-than or greater-than operator, and provide conversion between symbols and character strings.

4.5.2 Numbers

The following numeric types and operations are provided by all three languages: floating-point and integer numbers, the four arithmetic operations, modulus, and comparison. In OPS5, no other facilities are available; recall that OPS5 has a mechanism allowing the user to call a function or subroutine written in the implementation language. For any numeric manipulation beyond these simple facilities, this is the only method.

Prolog, in addition to the above, provides a few other operations, including exponentiation, square root, and some trigonometric and logarithmic functions.

Lisp has by far the richest set of features. Data types include up to four sizes of floating-point numbers, integers of arbitrary size, complex numbers, and true rational numbers. It has a very complete set of operations, including all those mentioned for Prolog, plus an odd/even test, maximum and minimum, gcd and lcm, hyperbolic functions, and a random number generator, among others.

4.5.3 Characters

While most conventional languages provide many features in support of character strings, KBS languages tend to place less importance on such facilities. OPS5 does not have character strings at all. Prolog represents character strings as a list of integers, each equivalent to the ASCII character code. Since lists are inherently variable-length, so are strings. The only operations provided are simple I/O and comparison.

Lisp, as usual, has a very powerful set of features. Strings are represented as vectors (one-dimensional arrays) of characters, but simulate variable-length strings with a so-called fill pointer. Among the operations provided are:

1. Input and output
2. Comparison
3. Tests for alphabetic or numeric
4. Conversion from and to integers, based either on character code, or on character representation of a number of any radix
5. Case (upper and lower) testing and conversion
6. Trimming specified characters from left or right side of string

In addition to these operations which apply specifically to character strings, Lisp has operations which may be done on sequences (see section 4.6.1, below) of any type of element, strings being a special case.

4.5.4 Logical

Logical data represents two values, "true" and "false", and can be tested by control structures of a program. Both Lisp and Prolog have logical constants and provide the operations of conjunction, disjunction, and negation to compute new truth values. OPS5 does not have logical data.

4.5.5 Bit

Bit data is capable of taking on only two values, 0 and 1. Normally, it is implemented directly on the hardware bit structure of the underlying machine. We must anticipate the discussion of aggregates and explain how collections of bits are treated, as well as individual bit data items.

OPS5 does not support the bit data type. Prolog does not have single bits as a type of data, but is capable of treating an integer as a fixed-length (as determined by the implementation) string of bits. There are five bitwise operations provided: conjunction, disjunction, negation, left shift, and right shift.

In Lisp, bits can be aggregated in two ways, either in a bit-array, or treated as a 2's complement integer. Either way provides a variable-length collection. Both approaches allow many of the same operations: conjunction, disjunction, exclusive-or, equivalence, nand, nor, and other binary operations. Both support unary negation, counting significant bits (which may be zero-bits for negative integers), and fetching and testing single bits.

In the array approach, one can set individual bits, which is not directly supported by integers. Furthermore, besides these bit-oriented operations, bit-arrays inherit the generic capabilities of sequences, see section 4.6.1, below. On the other hand, integers allow testing with a mask, left and right shifting, and a boolean function to perform all 16 logical binary bitwise operations (of course, many of these are provided directly, as mentioned above).

4.6 Aggregate Data

Aggregation is any mechanism in the language for building up a data structure from smaller pieces, such as smaller aggregates or elementary data, together with the operations provided to manipulate these structures.

4.6.1 Lists and Arrays

Perhaps the two most immediately apparent differences between KBS languages and conventional languages are that the former support symbols (see section 4.5.1) as an elementary data type, and *lists* as an aggregate type. From a sufficiently abstract

viewpoint, lists and one-dimensional arrays are the same. They are both linear *sequences* of objects. Certain distinctive characteristics, however, both logical and physical, are usually associated with each. Table 7 describes the differences. The entries in the table are not hard and fast rules, but are typical of the way many languages treat these structures.

Table 7: Characteristics of Lists and Arrays

Lists	Arrays
Direct access to first element only	Direct access to any element
Implicitly numbered positions	Explicitly numbered positions
Change by insertion or deletion	Change by replacement
May grow or shrink	Size is static
Elements of different type	Homogeneous Elements
One-dimensional	May be multi-dimensional
Implemented by linked pointers	Implemented by sequential storage

Prolog has lists which behave much as described in the table. The only conventional operations directly available are storing and fetching the first element of the list, conversion between lists and compound terms (where the first list element is taken as the functor, and the remaining elements as the arguments), and sorting. Of course, lists may participate in pattern matching, as described earlier.

OPS5 has what its documentation refers to as lists, but in many respects are more like arrays. Although their source code representation may be in the style of lists, they are implemented as arrays. Each member of the list has a unique identifying number, and fields are stored sequentially, from the end of an attribute-value element, or as an independent working memory element. The index number may be used explicitly. The maximum size is static and any element may be accessed or replaced directly. Unlike conventional arrays, however, the contents need not be of the same type (although they must be atomic).

Lisp has lists and arrays. It treats both lists and one-dimensional arrays as special cases of a more general type, called sequences. The following operations are provided for all sequences, although some may be more efficient for a list or an array.

1. Fetching and storing positionally specified elements and subsequences
2. Concatenation
3. Reversing order of sequence
4. Finding the length of the sequence

5. Initializing a subsequence to a single value
6. Various kinds of searching for matching subsequences and elements
7. Element translation within a sequence
8. Deleting specified elements from a sequence
9. Deleting duplicate elements
10. Testing whether all, some, or none of the elements meets some condition
11. Mapping, i.e., forming a new sequence, each of whose elements are a function of the corresponding element of the original sequence
12. Reduction, i.e., applying a function to all the elements as operands, to produce one result
13. Counting occurrences of a specified element
14. Sorting and merging sequences

Lisp is the only language with multi-dimensional arrays. Aside from the usual features described in table 7, Lisp arrays allow heterogeneous elements, and may be adjustable in size.

In addition to the sequence operations, Lisp can treat lists as sets, see section 4.6.3, below. Also, Lisp supports a so-called association list, in which each element consists of a key and some associated data. Elements can then be accessed by the key value. Since the association list is a list, it is searched linearly for a key match. Only the first match is returned if there are duplicate keys.

Finally, Lisp provides an aggregate called a hash table, which is somewhat like an association list, in that it allows access to data based on the value of a key. The difference is that hash tables allow only one entry per key, and are intended to be implemented by hashing, rather than with linear search.

4.6.2 Records

In conventional languages, records are fixed-format aggregates of logically related, but heterogeneous data, located in named fields. In section 3.1.3 earlier, we described the correspondence between the way KBS languages represent facts, and conventional records.

Prolog's *compound terms,* or structures, are its closest approximation to records. Within the compound term, however, fields are distinguished by position alone; there is no associated name.

OPS5's *attribute-value elements* correspond quite closely to the usual concept of records. The major difference, as mentioned earlier, is that the contents of a field must be atomic, not another structured object. A surprising point about implementation, however, is that if the same field name is used in two different record types, that field will occupy the same position in both underlying arrays. Thus, fields in a record may not be stored sequentially, and furthermore, this may affect the behavior of the program, since some operations depend on field order.

Lisp *structures* correspond almost exactly to conventional records. The major difference is that, as usual for Lisp, the contents of a field (called a *slot*) may be of any data type. However, it is also possible to declare a field to be of a certain type. The declaration of a Lisp structure implies the creation of functions to access the fields of the record, a named data type for the record, constructor and copy functions for records of that type, and other facilities.

4.6.3 Sets

A set of objects is an unordered collection. Any object must either belong to the set, or not. Typical operations on sets include adding or deleting a member, testing for membership, taking the union, intersection, or negation of sets.

Although both Prolog and OPS5 have lists, they have no built-in operations for treating these as sets. Prolog does have a unique facility, called *setof*, which, given a general specification, builds a list of all the objects conforming to that specification. This is one of the few cases where a language supports the construction of a set, not by an explicit listing of its members, but rather according to a rule.

Lisp can treat lists as sets, so that one can represent sets of arbitrary Lisp objects. The operations provided include union, intersection, set difference, exclusive-or, adding or deleting a member, membership test, and subset test. In addition, sets of non-negative integers can be simulated rather directly by the aggregates of bits described above in section 4.5.5.

4.6.4 Files and I/O

All the languages possess some means of displaying or accepting information from an external source, either a user's interactive terminal or relatively permanent file. Note that all but the simplest of KBS's rely on I/O facilities to construct a customized interface to the program for the end-user. In general, Lisp has the most elaborate facilities of the three languages, OPS5 the simplest. Table 8 summarizes the languages' I/O features. See section 3.1.5 for KBS-oriented I/O facilities.

Table 8: Files and I/O Features

Features	Lisp	Prolog	OPS5
File types	Sequential and Random	Sequential	Sequential
I/O language objects	Yes	Yes	Yes
I/O characters	Yes	Yes	
I/O binary	Yes		
Line input	as characters		as language objects
Pseudo-I/O	Yes		
User control of input	Many features	Some	
User control of output	Many features	Some	Few
Rename and delete files	Yes	Yes	

Lisp supports not only sequential files, but random-access files, which are addressed according to an integer position. Files can be input or output or bidirectional. Also, Lisp has pseudo-I/O, in which the source and destination of data is a character string data item. This allows the programmer to make use of the I/O system's parsing facilities to manipulate data internally. Lisp can perform I/O either on arbitrary Lisp objects, or at the character level. Also, an entire line of input can be accepted as a character string. There are some simple operations to perform binary I/O. Finally, files can be renamed or deleted.

In Lisp the user has a great deal of control over the parsing and interpretation of input and generation of output (format control). For instance, special characters in an input stream can cause a function to be invoked to evaluate the following expression. On output, Lisp's *format* function provides a host of facilities for controlling the presentation of data items, lines and pages.

Prolog has only sequential files. It does not have pseudo-I/O, line input, or binary I/O. Prolog can do I/O either at the character level or on Prolog terms (the equivalent of Lisp objects). Its facilities for formatting output include the ability to generate a new line or a tab, and an escape routine to a user program to control output.

OPS5 performs I/O on only language terms, not at the character level. A line of terms can be accepted as input. It has a few operations for output formatting, such as producing a new line, tabbing, and right justification.

4.6.5 Executable Code

Another distinguishing trait of KBS languages, aside from symbols, is the ability to treat sections of executable code as data. Code can be read in, written out, passed

as a parameter, examined, modified, and then, when appropriate, executed just as if it were part of the original source code. Only FORTRAN, among conventional languages, provides anything similar: it allows functions to be passed as parameters.

This capability supports the implementation of a number of useful techniques:

1. Control-oriented subroutines, in which a useful control structure (such as mapping of lists) is applied using any desired function or predicate.

2. Dynamic creation of rules to be added to the KB (see section 3.1.5).

3. Higher-order logic, in which the predicate itself (as well as arguments) can be a variable (allowing, for example, a query such as: "What predicate(s) relate John to Mary?").

At the syntactic level, treating code as data is achieved by expressing procedures in the same form as ordinary data objects. Thus, procedures (rules or functions) are expressed as lists in Lisp and OPS5, and as predicate clauses in Prolog. Since the processors for these languages are at least partially interpreters, they can execute whatever structure the program has built.

4.7 Macros and Pre-processing

Prolog can define one- or two-argument *operators*, which are then transformed into predicates. The operators can be infix, prefix or postfix. The user can control operator associativity and precedence. Also, Prolog has an evaluable predicate which allows the user to transform an input term according to any specified procedure before being loaded into the Prolog KB. These are both forms of pre-processing.

Lisp has very elaborate facilities for defining macros. The programmer can set up a template of a Lisp function or object, parameterized so that certain parts are filled in as specified in the macro invocation. Note that a Lisp macro is not just a static syntactic transformation, as with Prolog's pre-processing, but rather represents, conceptually, a run-time creation of source code, whose very structure may depend on the parameters with which it is invoked.

4.8 Miscellaneous Language Features

In this section we shall discuss briefly other important aspects of the languages which do not fall under any of the above general categories. As mentioned in the introduction, it is not the purpose of this report to cover every detail of each language, and therefore many minor features are not included here.

Lisp has a number of facilities worth mentioning:

1. Lisp has a non-local abort, in which the user can designate a section of code whose execution can be stopped immediately, even if control is not lexically within that section (perhaps because it invoked some other function).

2. Functions can return multiple values, as well as a single value.

3. There is a large assortment of functions to supply information about the environment (e.g., time and date) and implementation limits (e.g., numeric precision).

4. There are some functions for the manipulation of bytes as a data type.

Prolog's only additional feature is to allow the user to save the state of the current execution, exit completely from Prolog, and restore the saved state at some later time.

4.9 Simplicity

The simplicity of a language is a somewhat subjective property. We shall rely on the intuitive notion that a language with relatively few semantic concepts and syntax rules is simpler than one with more sophisticated structure. Clearly there is a trade-off between simplicity and power—ideally the user wants a language just powerful enough for the application, but with no superfluous complexity.

Prolog is the simplest of the three languages under study. It has a single model, the clause, for representing facts, rules, and queries. Its algorithm for resolving queries is a function of the lexical ordering of the source code. It offers a few simple data types.

OPS5 is also a rather simple language. It has only a few constructs for representing facts and rules, although these sometimes interact in subtle ways, as with the use of similar field names in different attribute-value elements. Its execution strategy, however, is rather sophisticated, and flow of control is often difficult to follow.

Lisp is by far the most complex of the three languages. It offers a wealth of data and control structures, and employs some very advanced computational concepts. Although flow of control is fairly straightforward, the effect of many of the functions can be quite subtle. For example Lisp has many pairs of functions which perform the same logical operation, but one of the pair changes its arguments (i.e., has side effects) and the other simply returns a value without such changes.

4.10 Standardization

Clearly standardization is an extrinsic characteristic of the languages—it does not depend on the language definition, but rather on external events. As of the time of this report (1987), there is no formally approved standard for any of the languages.

Of the three, only Lisp has been the subject of a successful conscious effort to produce a standard technical specification [Stee84]. Common Lisp seems to be gaining wide acceptance. In September 1986, technical commmittee X3J13 held its first meeting, under the auspices of the X3 committee (which is accredited by the American National Standards Institute), for the purpose of developing a formal Common Lisp standard. There is also a proposal under consideration for standardization of Lisp within the International Standards Organization (ISO). Nonetheless, there are still a great variety of dialects in use.

Unfortunately for such a young language, there are already divergent implementations of Prolog. The C-Prolog dialect [Pere84] is probably dominant, but by no means universal. Most of the divergence is at a relatively superficial syntax level, but there are some semantic problems as well, e.g., with the *cut* predicate. There is a Prolog panel currently working on a draft standard, under the auspices of the British Standards Institution (BSI). The results of this work would likely be proposed as an ISO standard.

The original manual on OPS5 [Forg81] serves as an informal *de facto* standard for the language. The effects of some language features, however, are explicitly described as being dependent on the implementation language or environment. There are no current plans for formal standardization.

4.11 Software Availability

It can be advantageous to be able to procure existing software to perform needed functions, rather than having to develop code. Lisp probably offers the greatest variety of ready-made software. Much of this software is available from academic sources, since Lisp has been a research language for a long time. Commercial products are also to be found. A conspicuous example is the so-called *flavor* system, offered by some vendors in support of object-oriented programming in Lisp.

Prolog and OPS5 have far less in the way of existing software. There are a number of Prolog libraries and extensions available, usually from academic sources.

5 User Requirements

In sections 3 and 4 we have examined the features offered by each of the languages, both conventional and KBS-oriented. In this section we shall discuss common user requirements, and relate these requirements back to the language features. There is no simple correspondence between requirements and features; in general it is a many-to-many relationship. Furthermore, user requirements will normally be stated at a more abstract, functional level, while language features often involve technical detail.

It is unlikely that any one language will best fulfill all user requirements. The point of this section is to help users consider explicitly several potentially relevant criteria. Having performed this analysis, users can then judge which criteria are most important, and choose accordingly.

We will proceed from those requirements most closely bound up with the logical definition of the application in question, to those dealing with practical constraints on implementation.

5.1 Functional Operations

The most basic requirement is that the language be able to perform the operations involved in the application. If the application appears to be goal-driven, with heterogeneous data, Prolog may be a good choice. OPS5 is suited for data-driven applications, with the data in a somewhat predictable format. See section 2.2.4 for more guidance. Also note that Prolog can express both facts and rules as part of the permanent KB, while OPS5 is more rule-oriented.

Although Lisp can be made to represent facts or rules, it does not provide any built-in derivation. If you are willing to develop your own inference engine, it is worth noting that algorithms for forward-chaining tend to be more complex than those for Prolog-style backward-chaining, and so implementation with Lisp will be more difficult for data-driven applications.

As far as conventional features go, Lisp, of course, provides a far richer set of features than the others. If the application requires a lot of sophisticated numeric and string manipulation, Lisp will provide these ready-made. Note that in those cases where OPS5 is implemented in Lisp, it should be possible to invoke Lisp functions from OPS5, so that if you need both forward-chaining and many conventional facilities, a multi-language approach may work best.

5.2 End-user Interaction

All the languages provide some built-in facilities for interaction with the end-user, and features for programming a customized interface. OPS5 is probably the least powerful in this regard. Prolog has many built-in facilities, especially for queries of the KB, but Lisp has features for building a more sophisticated interface. Some implementations of the languages have enhancements which allow the end-user to interact with the program through graphics facilities.

5.3 Size And Complexity

Prolog and OPS5 are both more oriented towards small and medium size programs than to large programs. They have no mechanism for structuring code above the individual rule level. For smaller programs, however, they are likely to incur less programming overhead than Lisp.

The Lisp package feature supports the large-scale organization of software. Lisp also has built-in features to inspect and document code, which are valuable in the management of large programming efforts. Moreover, for complex algorithms, where fine control is needed, Lisp provides a great variety of control mechanisms. By contrast, Prolog and OPS5 have only a single model for control which may not easily express such algorithms.

5.4 Execution Efficiency

Execution efficiency depends, of course, largely on the host hardware and quality of the language processor. Language design does exert some influence, however. All the languages are oriented towards a somewhat interpretative, as opposed to compiled, implementation. They were not designed with efficiency as the main goal. Lisp, however, does have a compile function, and, as a somewhat lower-level language, is likely to offer some advantage in run-time efficiency. Also, in cases where a needed function is built into Lisp, but would have to be defined in Prolog or OPS5, there is likely to be an advantage for Lisp, in that built-in functions can normally be implemented more efficiently. Conversely, the very size of Common Lisp implies a large run-time support package which can absorb computing resources.

5.5 Reliability

The languages' support for reliable programming depends to a large extent on the size of the program in question. For small programs, Prolog or OPS5 is likely to be advantageous, because they are simpler languages than Lisp, and have a presumably correct

inference engine built in. For larger coding efforts, however, the Lisp facilities for code modularization become quite important and may outweigh the simplicity of the other languages.

Lisp's type declarations may allow its compiler to identify mistaken or suspect pieces of code. OPS5 is weak in this regard. Many OPS5 implementations simply translate attribute names to the corresponding numeric field position, without checking whether the name was declared within the class (record type) being examined. Thus, OPS5 foregoes some of the benefit to be derived from declaring the format of its represention of facts. Note that the positional representation of facts in Lisp lists and Prolog compound terms is also subject to error.

Finally, recalling the potential problems with non-monotonic reasoning (see section 3.4.5), Prolog is the only language whose inference engine can be used in such a way as to guarantee monotonicity.

5.6 Application Stability

If the application is likely to be stable once coded, Lisp's efficiency advantage may make it a better choice than Prolog or OPS5. For more volatile applications, all the languages can claim good support of software maintenance. Prolog and OPS5, as somewhat higher-level languages than Lisp, are at some advantage for exceptionally dynamic applications. OPS5 has the advantage that the order of rules is completely insignificant, and so rules may be freely added or deleted.

5.7 Timeframe

Prolog and OPS5 are more appropriate than Lisp for projects of short duration and low overhead. Since they stress power over flexibility, it is easier to get a program into operation than with Lisp. Construction of prototypes is an important example of this kind of use. Conversely, if a software product is likely to have a long life span, then Lisp may be a better choice, since the language itself is somewhat more standardized and stable, and there may be some advantage in execution efficiency.

5.8 Number Of Programmers

When a project involves the co-ordinated effort of several programmers, Lisp's facilities for documentation and modularization become useful. For single-programmer applications, the power of OPS5 and Prolog put them at an advantage.

5.9 Programmer Expertise

Prolog, as the simplest of the languages, is a good choice when the coding is to be done by a non-professional programmer, i.e, one whose main skill is not programming, but perhaps expertise in the KBS domain. For such end-user programming, Prolog provides a ready-made and reasonably comprehensible inference engine, together with a simple, unified model of knowledge representation. OPS5 shares many of these Prolog traits, although it is a bit more complex, both in its knowledge representation and derivation process.

Lisp is more of a professional's language. It has a host of very sophisticated and powerful facilities, with which one may build a KBS, but which require a good deal of expertise for their proper use. The programmer must design and implement the application's knowledge representation and inference engine, based on the relatively low-level structures provided by Lisp.

5.10 Portability

When an application is to be implemented on several kinds of machines or systems, official or at least *de facto* standardization becomes a crucial factor. Lisp (as described in [Stee84]) has the highest degree of portability of the three. Most new implementations are based on this Common Lisp specification, and it is likely to be the dominant Lisp dialect for the foreseeable future. OPS5 and Prolog, by contrast, tend to be highly implementation-dependent, Prolog perhaps a bit more so than OPS5.

5.11 Language Availability and Hardware Support

Let us first consider the traditional environment with one central medium- or large-scale machine. Given a project of any size, the cost of an additional language processor can usually be justified if the language offers substantial benefits over those already available. Sometimes a project includes the procurement of hardware, and in these cases, language availability on that machine should be an important factor in the procurement. All three languages have been implemented on at least some large- and medium-scale machines. Lisp is widely available, but not always the Common Lisp dialect. Both Prolog and OPS5 have achieved a moderate degree of availability in this environment, Prolog probably somewhat more than OPS5.

The advent of microcomputers poses some new issues in the decision about languages. It is more common for a project to involve hardware acquisition, as compared to the use of larger machines. The cost (per machine, at least) is likely to be relatively low; hence the choice of language can be a freer one. As with the procurement of large systems,

language availability for a proposed microcomputer should be carefully considered before purchase. Prolog, as the simplest of the languages, is most easily implemented in the microcomputer environment, and is readily available. There seems to be less support for OPS5. Common Lisp is taken as the basis for implementation in many new products, but some of these provide only a subset of the full language described in [Stee84]. Given the size of Common Lisp, this is understandable; the user should determine, however, which features are not supported and whether they are important to the application.

Lisp is also coming to be widely available on so-called Lisp machines or AI workstations. These are powerful processors whose hardware has been designed specifically to support the kind of symbolic operations discussed in this report. While they offer the user high performance, they also may be limited to these special applications, and may support only one user at a time. At any rate, these machines often provide a full implementation of Common Lisp, and often have a powerful version of Prolog as well. Again, the availability of OPS5 is less than that for Lisp or Prolog.

5.12 Compatibility With Existing Software

If a KBS application is to operate in isolation then compatibility is not a constraint. It may be, however, that the KBS interacts with other software, either by sharing data or by invoking routines in other languages. Since this is a highly implementation-dependent area, we cannot make any general assertions about the ability of specific languages to interact successfully. Users should determine, as part of the project plan, whether the proposed language processor meets the application's compatibility requirements.

6 Summary and Conclusions

In this section, we shall recapitulate much of the information presented earlier in the report. In particular, we shall summarize the languages' salient characteristics, including strengths and weaknesses, and discuss the extent to which Lisp, Prolog, and OPS5 support features distinctive to KBS programming. Finally, we shall present some general conclusions about the appropriate circumstances for the use of each language.

6.1 Individual Language Review

6.1.1 Lisp

Of the KBS languages we've covered, Lisp is closest to the conventional languages. Indeed, it could be used much as if it were PL/I or Ada, given its facilities for various types of data, and for building data and control structures. Of course, Lisp introduced many of the unconventional features we've discussed, such as symbols, lists, and the ability to treat code as data.

Strengths:

- Lisp provides a great deal of flexibility and fine control.
- Lisp *structures* represent facts in a way which provides documentation, reliability, and features to support object-oriented programming.
- Lisp *packages* support large-scale modularization of programs.
- Lisp fully supports data abstraction with features for user-defined data types.
- Lisp has a very rich set of built-in data types and operations, including those for numeric, character, and bit data, both as elementary items and as aggregates. Among the supported aggregates are sequences, multi-dimensional arrays, and indexed tables.
- In Lisp the user has a great deal of control over the parsing and interpretation of input and generation of output (format control).
- Lisp's facilities for defining macros allow users to set up sub-languages designed for the application.
- Lisp is the best standardized of the three languages.

Weaknesses:

- Lisp does not provide an inference engine.
- There is no built-in pattern matching capability.
- There is no direct support for the representation of rules.
- Distinguishing between symbolic constants and variables may be awkward in Lisp.
- Lisp's built-in user interface is the least powerful of the three languages.

6.1.2 Prolog

Prolog was one of the first languages designed to apply inference rules to a set of premises so as to derive results automatically. It is based on standard predicate calculus as a means of representing information. Even though small and simple, it is a versatile tool for the implementation of a KBS.

Strengths:

- Prolog embodies a single, simple, reasonably efficient model for representing facts, rules, and queries, and for logical derivation.
- Prolog deals well with structured objects; its pattern matcher can easily compose and analyze lists and predicates.
- The built-in user interface is well-designed and has many useful features.
- The *setof* feature allows the user to gather up all solutions at once, rather than the more limited one-at-a-time approach.
- Prolog's predicates can be written so as to instantiate different arguments upon different invocations.

Weaknesses:

- Prolog's KBS model does not isolate logic from procedural control; the order of rules and facts, and of clauses within rules, may affect the results of derivation, making development and maintenance more complex.

- There is no mechanism for structuring or modularization of code on a scale larger than the single rule.
- Prolog allows only a single predicate in its then-part.
- The distinction between testing in the if-part of a rule, and asserting facts in the then-part is somewhat blurred, in that explicit updates of the KB are performed as a side-effect of a predicate in the if-part of a rule, and some testing may be implicit in the then-part.

6.1.3 OPS5

OPS5 is the result of an evolutionary process of language design to support efficient forward-chaining. Much of the structure of the language is determined by these efficiency requirements. Thus, OPS5 was originally motivated by more practical concerns than Prolog, whose impetus was somewhat more theoretical.

Strengths:

- OPS5 is highly declarative. The order of KB objects has no logical effect, and strategy is declared with a single command or statement.
- OPS5 has a sophisticated default control strategy.
- OPS5 can directly modify a fact by changing the values of some of its arguments.

Weaknesses:

- There is no mechanism for structuring or modularization of code on a scale larger than the single rule.
- The OPS5 model for a permanent KB provides only rules, not facts.
- OPS5 doesn't support nested data structures or recursively-defined data structures, in either its representation or pattern-matching facilities. Argument values must generally be atomic.
- There is no direct support for recursive procedures.
- The if-part of rules is rather constrained. No nesting of boolean operations is provided, nor does OPS5 support simple disjunctions at the predicate level.

- Computed expressions are not allowed in the if-part of rules.
- Somewhat arbitrarily, only a user command can delete a rule during a session, and only a program statement can add one.
- Even though the names of argument roles are declared, many implementations fail to warn the user of potentially mistaken references.

6.2 Summary Comparison of KBS Languages

6.2.1 Support of KBS Techniques

We have seen a number of features which distinguish KBS languages from conventional programming languages. These features all support the implementation of KBS applications, at various levels of abstraction. Their relationship to KBS has been covered in sections 2 and 3. The following stand out as characteristic:

1. **Symbols** as a primitive data type to represent concepts
2. Heterogeneous **lists** as one of the main mechanisms for aggregation of data
3. **Dynamic data structures** which can be built up and nested during execution without the need for a declared format.
4. **Free variables** capable of binding to any type of data, or of being unbound to any value
5. **Pattern-matching** in which the patterns can express general conditions by which objects are classified
6. More reliance on **applicative and declarative semantics** than on lexical sequence, procedural semantics, and control structures.
7. Treating **code as data** in which procedures, such as functions and rules, are manipulable objects
8. A **KB model** for the representation of facts and rules of the application domain
9. **Logical derivation**, in which rules are applied to facts, to generate new facts

Table 9 describes the degree of built-in support provided by each of the languages for these features.

Table 9: Language Support for KBS Features

Features	Lisp	Prolog	OPS5
Symbols	Yes	Yes	Yes
Lists	Yes	Yes	Partial
Dynamic data structures	Yes	Yes	No
Free variables	Yes	Yes	Yes
Pattern matching	No	Yes	Yes
Declarative semantics	Partial	Partial	Yes
Code as data	Yes	Yes	Yes
KB model	Partial	Yes	Yes
Logical derivation	No	Yes	Yes

6.2.2 Support for User Requirements

As discussed in section 3.5, the languages are most strongly differentiated along the dimension of flexibility vs. power. The implication for users is that, other things being equal, Lisp is appropriate for large, long-lived projects, implemented by professional programmers. Prolog and OPS5 are more oriented towards smaller-scale applications, and to situations where ease of coding has priority over execution efficiency. The choice between Prolog and OPS5 should depend mainly on whether the problem requires Prolog's more flexible data structuring facilities, or if OPS5's more controlled approach is suitable. Also, users should consider whether the problem is more naturally solved with a backward- or forward-chaining strategy.

Bibliography

[Bobr86] Daniel G. Bobrow, Sanjay Mittal, and Mark J. Stefik, "Expert Systems: Perils and Promise", *Communications of the ACM 29,* 9 (Sep. 1986), 880-894.

[Brat86] Ivan Bratko, *Prolog Programming for Artificial Intelligence,* Addison-Wesley, Reading, MA, 1986.

[Brow85] Lee Brownston, Robert Farrell, Elaine Kant, Nancy Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming,* Addison-Wesley, Reading, MA, 1985.

[Char80] Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott, *Artificial Intelligence Programming,* Lawrence Erlbaum Associates, Hillsdale, NJ, 1980.

[Clar84] K. L. Clark and F. G. McCabe, *micro-PROLOG: Programming in Logic*, Prentice-Hall, Englewood Cliffs, NJ, 1984.

[Cloc84] W. F. Clocksin and C. S. Mellish, *Programming in Prolog,* Springer-Verlag, New York, NY, 1984.

[Cohe85] Jacques Cohen, "Describing Prolog by its Interpretation and Compilation", *Communications of the ACM 28,* 12 (Dec. 1985), 1311-1325.

[Colm85] Alain Colmerauer, "Prolog in 10 Figures", *Communications of the ACM 28,* 12 (Dec. 1985), 1296-1310.

[Cugi84] John V. Cugini, *Selection and Use of General-Purpose Programming Languages,* NBS Special Publication 500-117, National Bureau of Standards, Gaithersburg, MD, 1984.

[Epst86] Jonathan A. Epstein, "Intelligently Speaking", *Digital Review* (May 1986), 70-78. Lists implementations of Lisp, Prolog, and OPS5 available for DEC computers.

[Fike85] Richard Fikes and Tom Kehler, "The Role of Frame-Based Representation in Reasoning", *Communications of the ACM 28,* 9 (Sep. 1985), 904-920.

[Forg81] Charles L. Forgy, *OPS5 User's Manual,* CMU-CS-81-135, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[Forg82] Charles L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", *Artificial Intelligence 19,* (1982) 17-27.

[Gene85] Michael R. Genesereth and Matthew L. Ginsberg, "Logic Programming", *Communications of the ACM 28,* 9 (Sep. 1985), 933-941.

[Gris71] R. E. Griswold, J. F. Poage, I. P. Polonsky, *The SNOBOL4 Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

[Haye83] Frederick Hayes-Roth, D. A. Waterman, and D. B. Lenat, *Building Expert Systems*, Addison-Wesley, Reading, MA, 1983.

[Haye84a] Frederick Hayes-Roth, "The Knowledge-Based Expert System: A Tutorial", *Computer 17,* 9 (Sep. 84), 11-28.

[Haye84b] Frederick Hayes-Roth, "Knowledge-Based Expert Systems", *Computer 17,* 10 (Oct. 84), 263-273.

[Haye85] Frederick Hayes-Roth, "Rule-Based Systems", *Communications of the ACM 28,* 9 (Sep. 1985), 921-932.

[IEEE83] *Computer 16,* 10 (Oct. 83). This issue is devoted to articles on knowledge representation.

[Pere84] Fernando Pereira (ed.), *C-Prolog User's Manual,* University of Edinburgh, Edinburgh, Scotland, 1984.

[Stee84] Guy L. Steele Jr., *Common Lisp,* Digital Press, Burlington, MA, 1984.

[Subr85] P. A. Subrahmanyam, "The 'Software Engineering' of Expert Systems: Is Prolog Appropriate?", *IEEE Transactions on Software Engineering SE-11,* 11 (Nov. 1985), 1391-1400.

[Wins84a] Patrick Henry Winston and Berthold Klaus Paul Horn, *Lisp*, Addison-Wesley, Reading, MA, 1984.

[Wins84b] Patrick Henry Winston *Artificial Intelligence*, Addison-Wesley, Reading, MA, 1984.

Glossary

Antecedent - See **If-part**.

Application - The **application** of a general rule to a set of facts in order to generate new facts is the basic operation of the inference engine in the construction of a derivation.

Applicative - In **applicative** programming, control is exercised mainly through the application of a function to its arguments, rather than through sequential flow of control. Ideally, the function simply returns a value, without side-effects.

Argument - The **arguments** of a predicate are those entities described or related by the predicate.

Attribute - See **Role**.

Backward-chaining - In **backward-chaining**, the derivation is constructed starting at the goal, and working towards the given facts.

Binding - The **binding** of a variable, typically as part of pattern-matching, means either that the variable is assigned a definite value, or that it is associated with another variable, so that they must both ultimately be assigned the same value.

Command - An instruction issued directly by the user of a KBS as part of his interaction with the system, cf. **statement**.

Consequent - See **Then-part**.

Constant - An element of code which always represents the same self-evident value, cf. **variable**.

Constraint - A general condition asserted about a knowledge base, forbidding or mandating certain kinds of facts or combinations of facts.

Control knowledge - Any knowledge concerning the order in which several applicable rules should be applied.

Data-driven - See **forward-chaining**.

Declarative - In **declarative** programming, control is implicit, statements express fixed facts or rules, and the order of statements is immaterial.

Derivation - In a **derivation**, new facts are deduced from facts already known, based on the application of general rules, following the normal conventions for proofs in logic.

Domain - The realm of knowledge within which a KBS operates.

Expert system - Software whose main purpose is to simulate expertise in some domain.

Fact - A true statement about the properties of or relationships among some specific object(s) expresses a **fact**, cf. **rule**.

Forward-chaining - In **forward-chaining**, the derivation is constructed starting at the given facts, and working towards the goal(s).

Goal - The fact(s) resulting from a derivation which constitute an answer to a user's query.

Goal-driven - See **backward-chaining**.

If-part - The part of a rule in which the current state of the knowledge base and derivation are examined, typically through pattern-matching, to see if the rule is applicable.

Inference engine - The active process which constructs a derivation in response to a query, drawing on the knowledge base.

Knowledge base - The set of given facts and general rules available to the inference engine for construction of the derivation.

Knowledge-based system - A system in which knowledge is encoded as facts and rules, and in which new facts are derived by applying the rules.

List - An ordered collection of arbitrary objects.

Monotonic - In monotonic reasoning, derived facts never become false, nor do disproven facts ever become true at a later stage in the derivation.

Object-oriented - A type of knowledge representation in which all the properties of an object are represented together in one predicate, cf. **property-oriented**.

Pattern - A template-like general description of a type of object.

Pattern matching - The process of determining whether an object falls within the category defined by a pattern, and what bindings of variables in the pattern are implied.

Predicate - A property or relationship which holds of or among an object or set of objects, represented by its arguments.

Procedural - In **procedural** programming, control is exercised by the order of statements and by explicit structures to alter sequential execution.

Property-oriented - A type of knowledge representation in which each property of an object is represented by a separate predicate, cf. **object-oriented**.

Query - A message from the user to a KBS requesting some information.

Response - A message from a KBS to the user providing requested information.

Role - The relationship of a particular argument to its predicate.

Rule - A general statement about the properties of or relationships among *kinds* of objects. Consists of an if-part and a then-part, cf. **fact**.

Rule base - That part of the knowledge base containing rules, as opposed to facts.

Statement - An instruction within a program, cf. **command**.

Strategy - The **strategy** of an inference engine is whatever method it uses to decide the order in which to construct the derivation.

Symbol - A type of atomic value within a KBS. Symbols are used to represent objects or concepts of the domain.

Then-part - The part of a rule which expresses the new facts implied by the the truth of the if-part.

Variable - An element of code which may take on different values at different times, especially within a rule, in which it takes on different values for different applications, cf. **constant**.

* U.S.Government Printing Office: 1987 0-181-076 (40220)

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBS/SP-500/145	2. Performing Organ. Report No.	3. Publication Date February 1987

4. TITLE AND SUBTITLE

Computer Science and Technology:
Programming Languages for Knowledge-Based Systems

5. AUTHOR(S)

John V. Cugini

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
Gaithersburg, MD 20899

7. Contract/Grant No.

8. Type of Report & Period Covered

Final

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

Same as item 6.

10. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 86-600602

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

Knowledge-Based Systems (KBS) represent a new software methodology which can broaden the scope of computer applications. When developing such software at the programming level, symbolic languages offer features to the programmer not provided by traditional procedural languages. The three most widespread symbolic languages are Lisp, Prolog, and OPS5. An abstract model for a basic KBS and associated terminology is described. This provides a framework for evaluation of the languages. There are several criteria by which one may assess the relative merits of these languages for a given knowledge-based application. Some are related to the languages' expressiveness for typical KBS techniques, others to the user's requirements. An extensive set of these criteria is discussed, and the languages are evaluated in light of them. While Lisp offers more features for general-purpose and symbolic computing, it does not offer direct support for the derivation process. OPS5 and Prolog have features especially designed for KBS, but lack many common general-purpose constructs.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

expert systems; knowledge-based systems; Lisp; OPS5; programming languages, Prolog.

13. AVAILABILITY

☒ Unlimited

☐ For Official Distribution. Do Not Release to NTIS

☒ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

☐ Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

79

15. Price

USCOMM-DC 6043-P80

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SCIENCE & TECHNOLOGY

Superintendent of Documents,
Government Printing Office,
Washington, DC 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Bureau of Standards Special Publication 500-.

Name _____

Company _____

Address _____

City _____ State _____ Zip Code _____

(Notification key N-503)

NBS Technical Publications

Periodical

Journal of Research—The Journal of Research of the National Bureau of Standards reports NBS research and development in those disciplines of the physical and engineering sciences in which the Bureau is active. These include physics, chemistry, engineering, mathematics, and computer sciences. Papers cover a broad range of subjects, with major emphasis on measurement methodology and the basic technology underlying standardization. Also included from time to time are survey articles on topics closely related to the Bureau's technical and scientific programs. Issued six times a year.

Nonperiodicals

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a worldwide program coordinated by NBS under the authority of the National Standard Data Act (Public Law 90-396). NOTE: The Journal of Physical and Chemical Reference Data (JPCRD) is published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements are available from ACS, 1155 Sixteenth St., NW, Washington, DC 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The standards establish nationally recognized requirements for products, and provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

Order the **above** *NBS publications from: Superintendent of Documents, Government Printing Office, Washington, DC 20402.*

Order the **following** *NBS publications—FIPS and NBSIR's—from the National Technical Information Service, Springfield, VA 22161.*

Federal Information Processing Standards Publications (FIPS PUB)—Publications in this series collectively constitute the Federal Information Processing Standards Register. The Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service, Springfield, VA 22161, in paper copy or microfiche form.

U.S. Department of Commerce
National Bureau of Standards
Gaithersburg, MD 20899